**FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO**


# WEBGRAF 2 – Tradutor web de grafos de especificação de controlo lógico em código de programação normalizado.

**Ricardo Simão da Rocha Garcês**

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Prof. António José Pessoa Magalhães (Professor Auxiliar)

26 de Julho de 2010

WEBGRAF 2 – Tradutor web de grafos de especificação de controlo lógico em código de programação normalizado.

**Ricardo Simão da Rocha Garcês**

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: António Augusto de Sousa (Professor Associado)

Vogal Externo: Jose Manuel de Castro Torres (Professor Auxiliar)

Orientador: António José Pessoa Magalhães (Professor Auxiliar)

_____

27 de Julho de 2010

# Resumo

A presente dissertação apresenta uma ferramenta de tradução para geração automática de um programa de controlo sequencial a partir do seu modelo comportamental definido numa rede de Petri ou em GRAFCET[1].

Esta ferramenta é uma aplicação web que recebe uma rede de Petri ou um grafcet, em XML, e retorna o ficheiro do programa PLC correspondente codificado numa das duas linguagens textuais definidas na norma IEC 61131-3: Texto Estruturado - Structured Text (ST) - e Lista de Instruções - Instruction List (IL). No caso das redes de Petri o formato XML usado para descrever as redes é o PNML, que está definido na norma ISO/IEC 15909-2:2009. No caso do grafcet, é usada uma proposta de explicitação textual, por não existir nenhum formato normalizado.

De forma a facilitar o acesso à aplicação, esta foi desenvolvida utilizando a tecnologia JavaServer Pages (JSP), o que permite a sua disponibilização na web. Outra característica da aplicação é a de permitir gerar código PLC normalizado. Actualmente, não é conhecida qulquer aplicação que faça geração de código a partir duma especificação em rede de Petri ou GRAFCET, com as características acima referidas.

1 De acordo com [IEC02], o termo GRAFCET denota a metodologia, e o termo grafcet denota um gráfico.

# Abstract

The dissertation presents a translation tool for the automatic generation of a sequential control program from its corresponding behavioural model defined in a Petri net or GRAFCET[1].

This tool is a web application that gets a Petri net or a grafcet described according to a XML standard format, and returns the corresponding PLC program file in a code compliant to the textual programming languages defined in the IEC 61131-3 standard: Structured Text or Instruction List.

PNML is the XML based format used to describe Petri nets, which is defined in the ISO/IEC 15909-2:2009 format. In the case of grafcet, it is proposed a textual description because presently there is no standard format.

In order to allow the access to the application worldwide, it was developed using JavaServer Pages (JSP) making it available on the web. Another feature of the application is that it generates PLC standard code. Currently, no other application generating programming code from Petri nets specifications exhibiting both features seems to exist.

1 According to [IEC02], GRAFCET denotes the methodology and grafcet denotes a chart.

# Agradecimentos

Expresso os meus agradecimentos a todos os que contribuíram para a elaboração deste trabalho, e em especial ao orientador Professor Doutor António José Pessoa de Magalhães pela sua total disponibilidade e colaboração.

Ricardo Simão da Rocha Garcês

# Índice

# Lista de Figuras

# Lista de Tabelas

# Abreviaturas e Símbolos

| | |
|---|---|
| DOM | Document Object Model |
| GRAFCET | GRAphe Fonctionnel de Commande Etape Transition |
| IL | Instruction List |
| JAXP | Java for XML Processing |
| JSP | JavaServer Pages |
| PLC | Programmable Logic Controller |
| PNML | Petri Net Markup Language |
| SFC | Sequential Functional Chart |
| SIPN | Signal Interpreted Petri Nets |
| ST | Structured Text |
| UML | Unified Modeling Language |
| XML | eXtensible Markup Language |

# 1 Introdução

Este capítulo apresenta os objectivos e motivações deste trabalho, bem como a organização de toda a dissertação.

## 1.1 Motivação e Objectivos

Nos últimos anos, e para responder às necessidades do mundo moderno, tem-se assistido a uma crescente necessidade de desenvolvimento de sistemas de eventos discretos cada vez mais exigentes e complexos. A maioria destes sistemas é aplicada em Controladores Lógicos Programáveis (PLCs) [Bol09]. Os exemplos incluem: controlo de instalações industriais, sistemas de produção automatizados e sistemas de controlo distribuídos, entre outros. Portanto, as soluções ad hoc e intuitivas usadas no passado já não são razoáveis nos dias de hoje. Actualmente, formalismos de modelação para projectar, testar e validar sistemas feitos pelo homem são absolutamente necessários, e as redes de Petri e o GRAFCET são particularmente relevantes neste contexto [CL99].

A teoria de redes de Petri foi criada por Carl Adam Petri em 1962 com o objectivo de permitir a modelação de sistemas de eventos discretos. As vantagens em usar modelos baseados em redes de Petri são numerosas, o seu grande poder de expressividade permite representar facilmente todas as relações de causalidade entre processos em situações de conflito, concorrência e sincronização, permite também a utilização de técnicas de análise, tais como a árvore de alcançabilidade e de cobertura, que assegura que um sistema está livre de erros [Mur89].

O GRAFCET é uma linguagem de especificação para modelação de sistemas sequenciais, e foi desenvolvida a partir das redes de Petri. Ou seja, ambas as linguagens podem ser aplicadas em sistemas sequenciais que evoluam discretamente e por eventos.

Ao longo dos últimos anos, tem-se sentido uma crescente utilização do GRAFCET e das redes de Petri na especificação comportamental dos controladores. Por outro lado, há uma maior utilização de Controladores Lógicos Programáveis (PLCs). Dito isto, faz sentido que o

desenvolvimento de sistemas de automação passe por uma tradução de um modelo de controlo especificado em GRAFCET ou redes de Petri para código de programação de PLCs, e, sendo assim, as ferramentas de software que permitem esta tradução, gerando código a partir de um modelo, são uma ajuda importante para os programadores de sistemas. No caso das ferramentas actuais geradoras de código a partir de um modelo especificado em redes de Petri, os seus problemas principais são adoptar um formato próprio para descrever as redes de Petri, e o código gerado ser dependente da plataforma ou específico de um determinado PLC. E, por estas razões, não estão a ser amplamente utilizadas.

A ferramenta apresentada nesta dissertação ultrapassa os problemas mencionados acima já que usa um formato normalizado para especificação das redes de Petri - Petri Net Language Markup (PNML). Como não existe um formato para especificação de GRAFCETs, foi proposto e usado um formato baseado em XML. Por outro lado também gera código normalizado de PLC.

A ferramenta, de nome WEBGRAF 2, é uma actualização à versão anterior, WEBGRAF [Gom03], que tinha os problemas já mencionados. Pretende-se que esta ferramenta seja gratuita, e, por isso, a Web é usada como plataforma de trabalho pois a Internet desempenha um papel fundamental a tornar qualquer ferramenta disponível a qualquer pessoa. É de referir que o interesse do trabalho que aqui se apresenta foi já reconhecido através da aceitação para apresentação, e publicação nas respectivas actas, de um artigo de divulgação científica num congresso internacional a realizar proximamente [GM10].

## 1.2 Estrutura da Dissertação

A dissertação está organizada em 7 capítulos. O segundo capítulo descreve o estado da arte, onde são apresentados trabalhos relacionados. O capítulo 3 apresenta uma breve descrição das redes de Petri, e apresenta o respectivo formato normalizado de especificação textual: PNML. No capítulo que se segue, capítulo 4, é feita uma apresentação do GRAFCET e da sua linguagem de especificação: GML. O capítulo 5 destina-se a apresentar a estrutura básica de um PLC e as linguagens de programação textuais geradas pela aplicação WEBGRAF 2: IL e ST. O capítulo 6 é responsável pela apresentação da arquitectura geral da aplicação e das tecnologias utilizadas no seu desenvolvimento. Também apresenta a arquitectura geral do código gerado pela aplicação e discute, individualmente, as diferentes unidades de código. No final do capítulo é dado um exemplo de utilização da aplicação. Finalmente, o capítulo 7 destina-se a apresentar as conclusões do trabalho e perspectivas de prossecução.

# 2 Revisão Bibliográfica

O presente capítulo destina-se a apresentar o estado da arte. Pretende-se verificar a existência de trabalhos relacionados e descobrir as suas falhas para se averiguar de que maneira é que a aplicação apresentada nesta dissertação as consegue colmatar.

A ferramenta tem como objectivo traduzir modelos de controlo sequencial em código de programação normalizado dos PLCs. A tradução tem como ponto de partida uma explicitação textual dos modelos, que terá que ser escolhido dentre os formatos de especificação existentes.

A criação do ficheiro da explicitação textual dos modelos de controlo é uma tarefa ingrata se feita através de um editor de texto. E portanto, pretende-se averiguar a existência de ferramentas que permitam a fácil criação destes ficheiros, nomeadamente, editores gráficos que suportam a exportação dos modelos para os seus respectivos formatos de especificação.

As secções seguintes irão abordar estes tópicos.

## 2.1 Trabalhos relacionados

Esta secção tem como objectivo dar a conhecer o estado da arte no que respeita a ferramentas que permitam a geração de programas de PLC a partir dum modelo de controlo sequencial. A ferramenta WEBGRAF, [Gome03], constitui o principal estado da arte, mas, fruto de pesquisas na Internet, verifica-se a existência de outras aplicações capazes de gerar automaticamente código de PLCs a partir duma especificação de redes de Petri. Estas ferramentas irão ser analisadas individualmente nas próximas sub-secções.

### 2.1.1 WEBGRAF

Como já foi anteriormente dito, a ferramenta WEBGRAF constitui o elemento principal do estado da arte. Tal razão deve-se ao facto de a ferramenta WEBGRAF 2 pretender ser uma actualização do WEBGRAF. Como tal, têm os mesmo objectivos - ser um tradutor web de

modelos de controlo sequencial em código de programação dos PLC. Contudo, possuem aspectos que os diferenciam. Para começar, veja-se algumas características do WEBGRAF:

- Utilização de um formato próprio para especificar redes de Petri;
- Utilização de um formato próprio para especificação de grafcets;
- Geração de código IL destinado a um tipo de PLCs (família Simatic S7-200 da SIEMENS).

Analise-se agora essas características:

- O facto de utilizar um formato próprio para especificar os modelos de controlo, dificulta a criação dos mesmos. Pois verifica-se a inexistência de editores gráficos de composição dos modelos com suporte à geração da respectiva explicitação textual.
- O facto do código gerado ter como alvo um tipo específico de PLCs, torna-o incompatível com os restantes.

A aplicação WEBGRAF 2 constitui uma melhoria destes aspectos, na medida em que pretende utilizar formatos, para especificação dos modelos, já utilizados por outras ferramentas, e de preferência que sejam formatos normalizados. Já o código gerado, pretende-se que seja normalizado para que tenha como alvo qualquer PLC, e não se restrinja a um tipo específico. Além disso, também se pretende o suporte a geração de código ST.

### 2.1.2 SIPN-Editor

SIPN-Editor, [FM00], [SIPN02], é uma ferramenta que permite a edição, visualização e simulação de redes de Petri do tipo SIPN (Signal Interpreted Petri Nets).

À semelhança com o WEBGRAF 2, permite:

- Tradução de um modelo de controlo para um programa PLC escrito na linguagem IL normalizada (IEC 61131-3).

No entanto, possui alguns inconvenientes:

- Usa um formato próprio, baseado em XML, para especificação das redes;
- Só são permitidas redes do tipo SIPN;
- Não é uma ferramenta Web.

Apesar de possuir um editor gráfico com suporte a simulação do modelo, a verdade é que, o facto de possuir um formato próprio de especificação, impossibilita a transferência do modelo para outras ferramentas com técnicas de análise mais poderosas, que é essencial ao desenvolvimento de sistemas de controlo complexos.

A aplicação WEBGRAF 2 permite a tradução de redes de Petri do tipo Lugar/Transição que são as mais comuns, e também algumas extensões. Além disso, permite também a tradução de grafcets.

A Figura 2.2 apresenta o aspecto gráfico desta ferramenta.

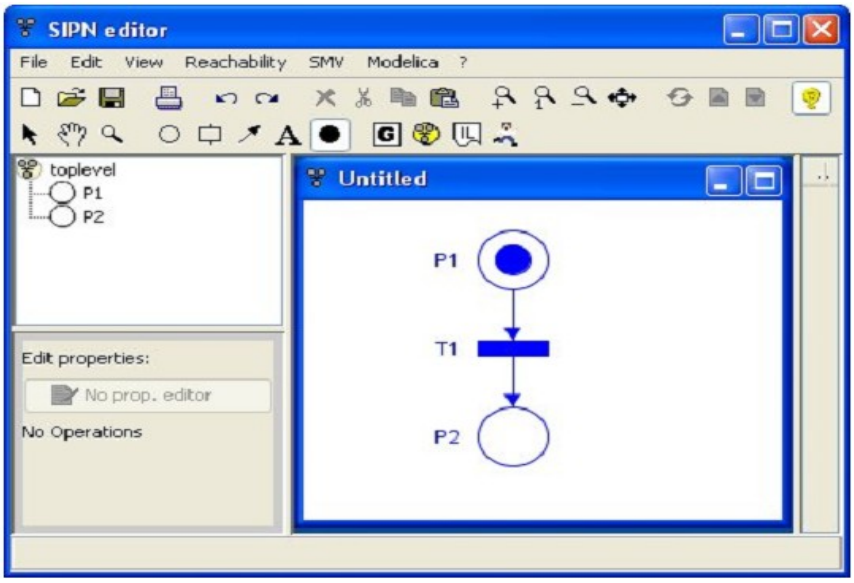



Figura 2.1: Aspecto gráfico da ferramenta SIPN Editor.

### 2.1.3 PetriLLD

A aplicação PetriLLD suporta a geração de algumas linguagens IEC 61131-3 (LD, ST e IL) a partir de uma linguagem gráfica baseada em redes de Petri. Usa um formato textual próprio, baseado em XML, para especificação dos modelos.

Os inconvenientes desta ferramenta, ou diferenças em relação ao WEBGRAF 2, são:

- O modelo é descrito numa linguagem própria da ferramenta, baseada em redes de Petri, que não tem muito uso;
- Não é uma ferramenta Web.

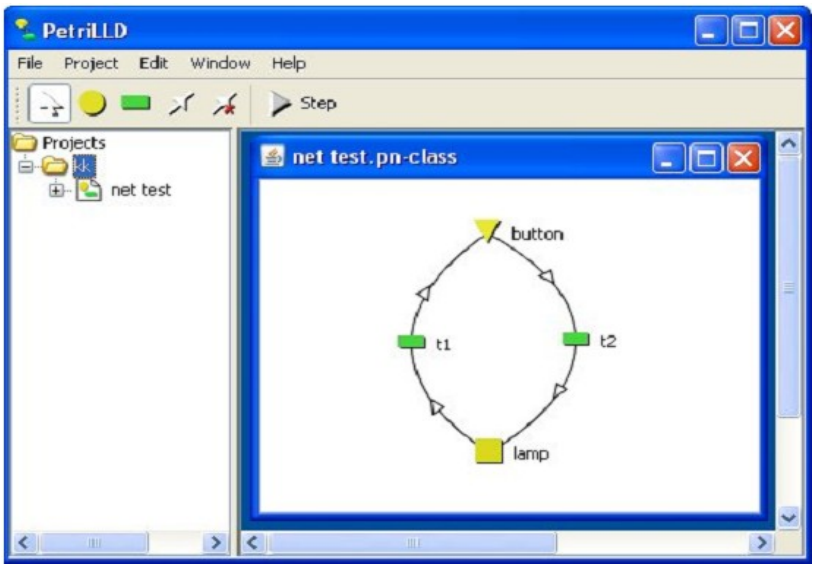



Figura 2.2: Aspecto gráfico da ferramenta PetriLLD.

## 2.2 Linguagens de especificação dos modelos de controlo sequencial

A presente secção, pretende dar a conhecer os formatos textuais existentes de especificação dos modelos de controlo sequencial, que possam servir como ponto de partida à geração dos programas de PLCs, por parte da aplicação apresentada nesta dissertação.

No contexto da especificação textual dos modelos de controlo sequencial destaca-se o PNML para especificação de redes de Petri. Relativamente ao GRAFCET, verifica-se a inexistência de um formato textual para especificação do mesmo. No entanto, verifica-se a existência de um formato textual para especificação de SFC (*Sequential Funtion Charts*), que poderá servir de ponto de partida para a criação de uma linguagem de especificação de GRAFCETs, visto ter vários elementos em comum com a mesma.

### 2.2.1 Linguagem de especificação de redes de Petri

Petri Net Markup Language (PNML) é um formato para especificação de redes de Petri baseado em XML que surgiu devido à necessidade de existir um formato normalizado para especificação de redes de Petri de forma a permitir intercâmbio de redes entre diferentes ferramentas de software. E está definido na Parte 2 da norma internacional ISO/IEC 15909 [ISO09].

### 2.2.2 Linguagem de especificação de GRAFCET

Até à data, não são conhecidos formatos textuais de especificação de grafcets. O que sugere que seja necessário criar uma proposta de formato de especificação de grafcets. No entanto, verifica-se a existência do formato de especificação PLCOpen XML [POX09], que é o formato de especificação das linguagens definidas na norma IEC 61131-3. Uma dessas linguagens é o SFC, que é uma linguagem de programação gráfica baseada em GRAFCET. Apesar de SFC e GRAFCET serem linguagens distintas (uma é uma linguagem de programação e a outra é uma linguagem de modelação) a verdade é que possuem vários elementos em comum. Portanto, o formato que se pretende criar, poderá ter com ponto de partida o PLCOpen XML, tentando aproveitar ao máximo a sua sintaxe.

## 2.3 Editores gráficos dos modelos de controlo sequencial com exportação para os formatos de especificação

No contexto da edição e simulação de redes de Petri, e exportação para o formato PNML, destaca-se a ferramenta JARP. Dentro das várias ferramentas existentes com estas características destaca-se esta devido a possuir uma interface gráfica bastante agradável e

apelativa, e também por ser uma ferramenta *opensource* multiplataforma (já que é desenvolvida em JAVA). A Figura 2.3 apresenta o aspecto gráfico desta ferramenta.

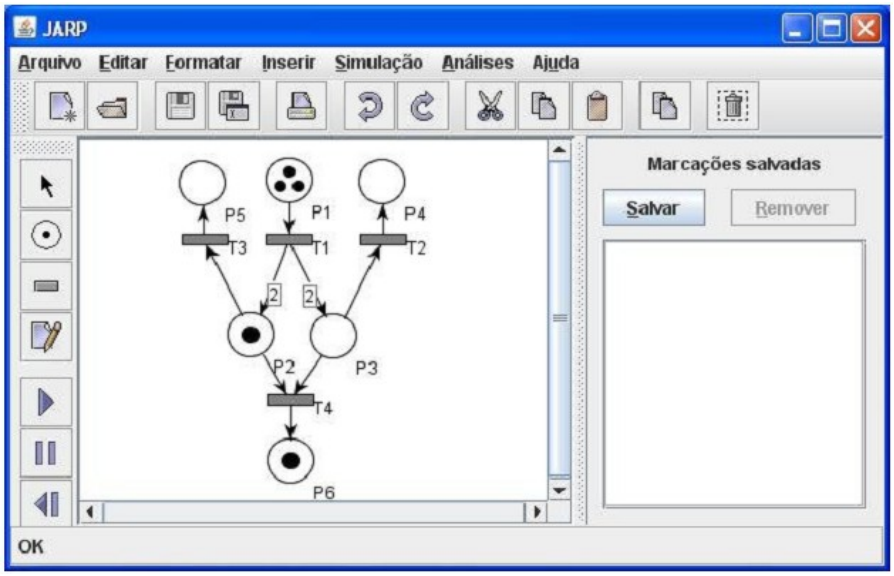



Figura 2.3: Aspecto gráfico da ferramenta JARP.

## 2.4 Resumo e Conclusões

Actualmente, verifica-se a existência de algumas ferramentas que permitem a geração de programas de controlo sequencial a partir dos seus respectivos modelos definidos em redes de Petri. Contudo, possuem limitações que fazem com que percam algum interesse. Nomeadamente, o facto de possuírem um formato próprio para especificação dos modelos de controlo sequencial, impossibilita a transferência dos modelos para outras ferramentas, ferramentas essas que podem possuir técnicas de análise poderosas a fim de testar e validar os modelos, o que nos dias de hoje é crucial no desenvolvimento de sistemas de controlo sequencial. O código gerado pela aplicação WEBGRAF tem como alvo uma família específica de PLCs e não possui um editor gráfico para geração do ficheiro com a explicitação textual do modelo, tornando o seu uso mais dificultado. O WEBGRAF é a única ferramenta conhecida com geração de código de PLC a partir dum modelo definido em GRAFCET, mas contém os problemas já mencionados anteriormente.

No que toca a redes de Petri, a aplicação WEBGRAF 2 consegue colmatar todas falhas do WEBGRAF, na medida em que é usado um formato normalizado para especificação das redes de Petri (o que permite o uso de ferramentas auxiliares para composição dos modelos e exportação para o seu formato de especificação - PNML), e o código gerado é normalizado (podendo executar em qualquer PLC ao invés de ter um único PLC como alvo). No que toca a GRAFCETs, não são conhecidas ferramentas de geração de código a partir de modelos desta linguagem e verifica-se a inexistência de um formato normalizado de especificação dos modelos. Optou-se, portanto, por criar um formato que tem por base o formato de especificação

de uma linguagem IEC 61131-3. No entanto, não existem editores gráficos que permitam a composição gráfica destes modelos e respectiva exportação para este formato de especificação.

# 3 Redes de Petri

As redes de Petri foram criadas por Carl Adam Petri, em 1962 na sua dissertação de Doutoramento. A sua aplicação inicial incidia sobre o estudo de autómatos. Desde então, devido ao seu potencial de modelação, numerosas extensões, trabalhos teóricos e aplicações têm vindo a ser desenvolvidos para responder às diferentes solicitações. As redes de Petri são um tipo de grafo bipartido, composto por três elementos principais: lugares, transições e arcos, e oferecem um ambiente para modelação de sistemas de eventos discretos, permitindo uma visualização global da sua estrutura e comportamento.

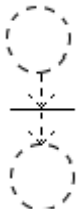
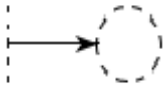
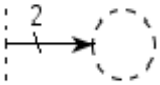
## 3.1 Noções básicas

Uma rede de Petri é composta por lugares, transições e arcos orientados. Um arco liga um lugar a uma transição ou vice-versa, nunca dois lugares ou duas transições. Um lugar que esteja ligado a uma transição por um arco que esteja orientado do lugar para a transição, é chamado de lugar de entrada dessa transição. Já se o arco estiver orientado da transição para o lugar, este chama-se lugar de saída. Os lugares podem conter marcas, e os arcos têm um peso, representado por um número natural.

## 3.2 Representação gráfica dos elementos

A Tabela 3.1 contém a representação gráfica de cada elemento constituinte de uma rede de Petri.

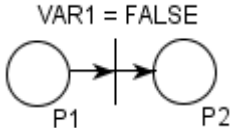
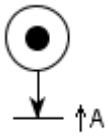
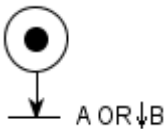
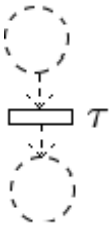
Tabela 3.1: Representação gráfica dos elementos de uma rede de Petri.

| Símbolo | Nome/Descrição | Exemplo | *[1] |
|---|---|---|---|
|  | **Lugar sem marcas:** Um lugar é representado por uma circunferência . * deverá ser substituído pela descrição do lugar. |   | X |
|  | **Lugar com marcas:** Um lugar pode conter um número inteiro de marcas, representadas por pontos negros. O exemplo 1 mostra um lugar (P1) com três marcas. O exemplo 2 mostra um lugar (P2) com uma marca. | Exemplo 1:   Exemplo 2:   | X |
|  | **Transição:** Uma transição é representada por um pequeno segmento de recta e é ligada por meio de arcos às transições de entrada e de saída. | | X |
|  | **Arco de peso unitário:** Um arco liga um lugar a uma transição, ou vice-versa. A orientação do arco deve ser indicada com uma seta. |  | X |
|  | **Arco de peso superior a 1:** Se o peso do arco for superior a 1, o arco deverá conter um pequeno traço atravessado com a indicação do seu peso. * deverá ser substituído por um número natural maior do que 1. |   Arco com peso 2. | X |

A Tabela 3.2 apresenta uma lista de extensões de redes de Petri aceites pelo tradutor.

1 A última coluna indica quais os elementos da tabela que estão contemplados na aplicação WEBGRAF 2. Estando esses marcados com uma cruz - 'X'.

Tabela 3.2: Representação gráfica das extensões das redes de Petri contempladas na aplicação WEBGRAF 2.

| Símbolo | Nome/Descrição | Exemplo | * |
|---|---|---|---|
|  | **Condição da transição:** A condição de uma transição é uma proposição lógica que pode tomar os valores “verdadeiro” e “falso”. A condição compreende uma ou várias variáveis. É necessário que a condição seja verdadeira para que a transição dispare. |  A condição da transição é verdadeira se VAR1 for falso. | X |
|  | **Flanco ascendente de uma variável lógica:** Um flanco ascendente associado a uma variável significa que o resultado dessa expressão só é verdadeiro quando a variável sofrer uma mudança de valor de 0 para 1. |  A condição da transição é verdadeira se A sofrer uma mudança de valor de 0 para 1 | X |
|  | **Flanco descendente de uma variável lógica:** Um flanco descendente associado a uma variável significa que o resultado dessa expressão só é verdadeiro quando a variável sofrer uma mudança de valor de 1 para 0. |  A condição da transição é verdadeira se A for verdadeiro ou B sofrer uma mudança de valor de 0 para 1. | X |
|  | **Transição temporizada:** Numa transição temporizada, a transição fica habilitada quando termina o tempo pré-estabelecido. | | X |

## 3.3 Condição de disparo de uma transição

Para uma transição poder disparar, todos os seus lugares de entrada têm que ter, pelos menos, um número de marcas igual ao peso dos arcos que os ligam à transição. No caso de possuir uma condição, é preciso também ela se verifique. Quando a transição é disparada, ela

consome as marcas necessárias e novas marcas são colocadas nos lugares de saída. Em cada lugar de saída o número de marcas criadas é igual ao peso do arco que o liga à transição.

As Figuras 2.3 e 3.2 mostram um exemplo de uma rede com uma transição a ser disparada: a Figura 2.3 representa o estado da rede antes do disparo; a Figura 3.2 representa o estado da rede após a transição ter sido disparada. O arco que liga o lugar de entrada $p_1$ à transição tem um peso unitário, e portanto é consumida uma marca desse lugar quando a transição é disparada. O arco que liga o lugar de entrada $p_2$ à transição tem um peso de 2, o que significa que duas marcas são consumidas aquando do disparo. Finalmente, aos lugares de saída $p_3$ e $p_4$ são adicionadas uma e duas marcas, respectivamente, quando a transição é disparada, que correspondem aos pesos dos respectivos arcos que os ligam à transição. Note-se ainda que para a transição disparar a sua condição - $var_1$ or $var_2$ - tem que ser respeitada, ou seja, pelo menos uma das variáveis ($var_1$ ou $var_2$) tem de ser verdadeira.

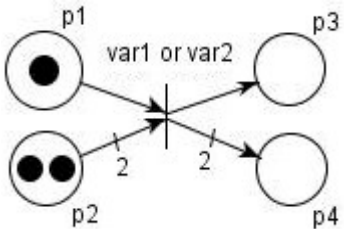



Figura 3.1: Rede de Petri pronta a ser disparada.

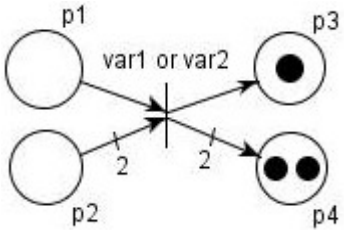



Figura 3.2: Rede da Figura 2.3 depois de a transição ter sido disparada.

## 3.4 Petri Net Markup Language (PNML)

Petri Net Markup Language (PNML) [PNML09] é um formato para especificação de redes de Petri baseado em XML [Har01]. Foi criado para permitir a especificação de todos os tipos de redes de Petri existentes, e foi desenvolvido de forma a ser extensível e aberto a futuras variantes das redes de Petri. Surgiu devido à necessidade de um formato normalizado para especificação de redes de Petri de forma a permitir o intercâmbio de redes entre diferentes ferramentas de software. Actualmente, o PNML está definido na Parte 2 da norma internacional ISO/IEC 15909 [ISO09] e define a sintaxe exacta de três versões de redes de Petri: Place/Transition Net, Symmetric Nets e High-Level Petri Net Graphs (HLPNGs), como definido na Parte 1 desta norma internacional [IEC04]. A Parte 1 contém a definição conceptual e matemática das três versões de redes de Petri mencionadas anteriormente. Finalmente, a Parte

3 irá conter a definição exacta do mecanismo de extensão de redes de Petri. A Parte 3 irá definir mais funcionalidades das redes de Petri: arcos inibidores, *reset arcs* e *read arcs*.

De forma a representar a definição exacta das redes de Petri em XML, PNML usa meta-modelos UML [UML10]. O PNML Core Model define os conceitos partilhados por todos os tipos de redes de Petri. Para além desse, o PNML ainda possui meta-modelos UML para especificar os três tipos de redes de Petri já mencionados: Place/Transition-Nets, High-level Petri Nets e Symmetric Nets. A Figura 3.3 ilustra os diferentes *packages* do PNML e as suas dependências.

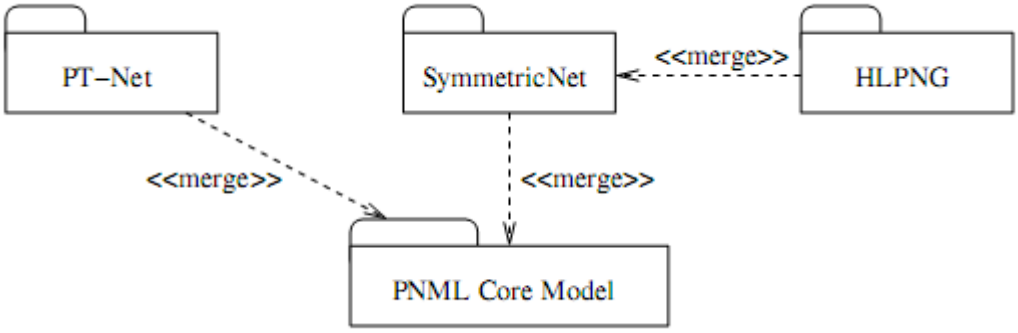



Figura 3.3: *Packages* do PNML

Nas sub-secções seguintes irão ser apresentados os conceitos e sintaxe XML dos *packages* PNML Core Model e Place/Transition-Net por estarem englobados no âmbito desta dissertação.

### 3.4.1 PNML Core Model

A Figura 3.4 ilustra o *package PNML Core Model* na forma de diagrama de classes UML.

*Petri net document* (PetriNetFile) é um documento que satisfaz os requisitos do *PNML Core Model*. Pode conter uma ou várias *Petri nets* (PetriNet) que, por sua vez, possui um *id* e um *type*. O *id* corresponde a um identificador único em todo o documento, o *type* corresponde a um URL referente ao nome do *package* com a definição da versão da rede de Petri. Uma *Petri net* é composta por uma ou várias *pages*, que por sua vez contêm *objects*. Os *objects* correspondem aos elementos gráficos das redes de Petri e também possuem um identificador único, *id*. Esses elementos são: *place*, *transition* e *arc*. Através da análise da figura anterior pode-se reparar que *place* e *transition* são generalizados como *node* que pode ser ligado por *arcs*. Note-se ainda que o *PNML Core Model* permite que *arcs* liguem dois *places* ou duas *transitions*. Cada *package* é que deverá impor a restrição de que transições ou lugares não podem ligar directamente entre si por um arco. Isto deve-se a poderem existir (ou poderem vir a existir) versões de redes de Petri que permitam que transições, ou lugares, estejam ligados entre si. Os *objects* podem conter *labels*. Existem dois tipos de *labels*: *annotation* e *attribute*. Uma *annotation* corresponde a qualquer informação textual do *object* correspondente, enquanto que *attribute* contém informação gráfica do *object*: tamanho, cor, etc. Note-se que as classes *label*, *annotation* e *attribute* são abstratas, os *packages* contendo as versões das redes de Petri é que deverão definir as suas *labels* concretas. As *annotations* poderão ser nomes, número de marcas

iniciais, peso dos arcos, etc. Como já dito anteriormente, cada *Petri net* é constituída por *pages*, e nas *pages* é onde estão inseridos os *objects*. Para redes de dimensões muito grandes, estas poderão ser estruturadas com várias *pages*, e para um *arc* ligar a uma *transition* ou *place* de outra *page* são usados *reference places*(RefPlace) e *reference transitions*(RefTrans), que correspondem a referências para *places* ou *transitions* de outras *pages*. Finalmente, *tool specific information* (ToolInfo) pode estar associado a *objects* ou *labels* e contem informação sobre a rede que seja usada específicamente por uma ferramenta.

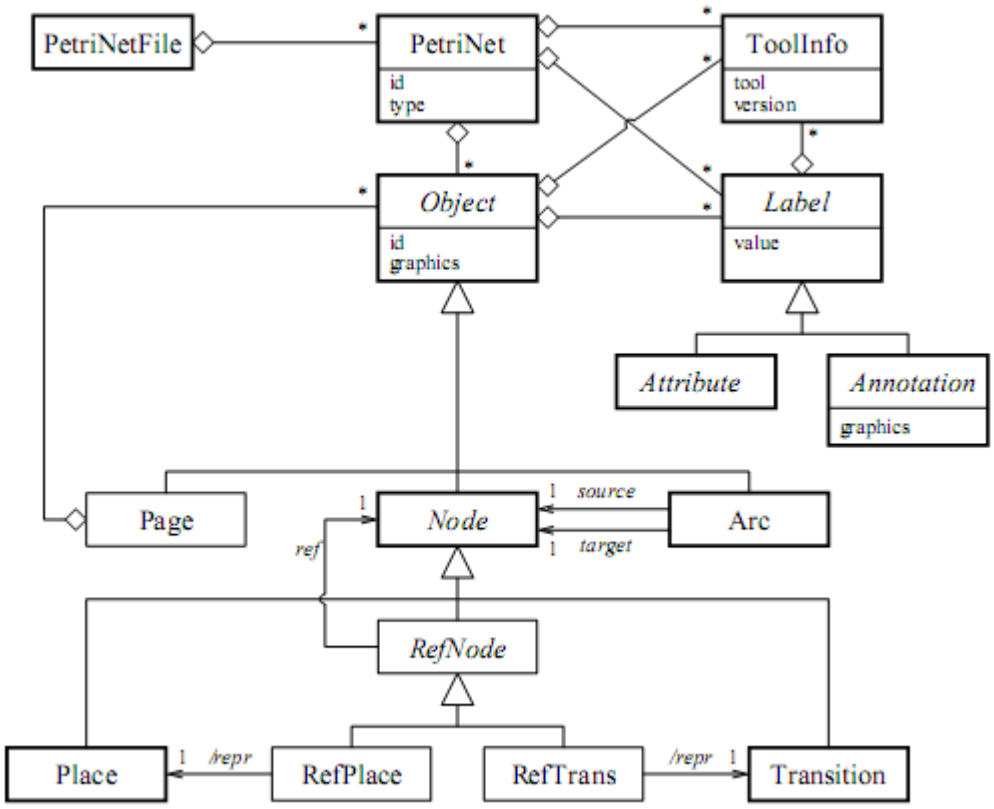



Figura 3.4: PNML Core Model ([BCHK03])

### 3.4.2 Place/Transition-Net

Irá agora ser discutido o meta modelo para as redes de Petri do tipo *P/T-Nets*, que se encontra ilustrado na figura 5.

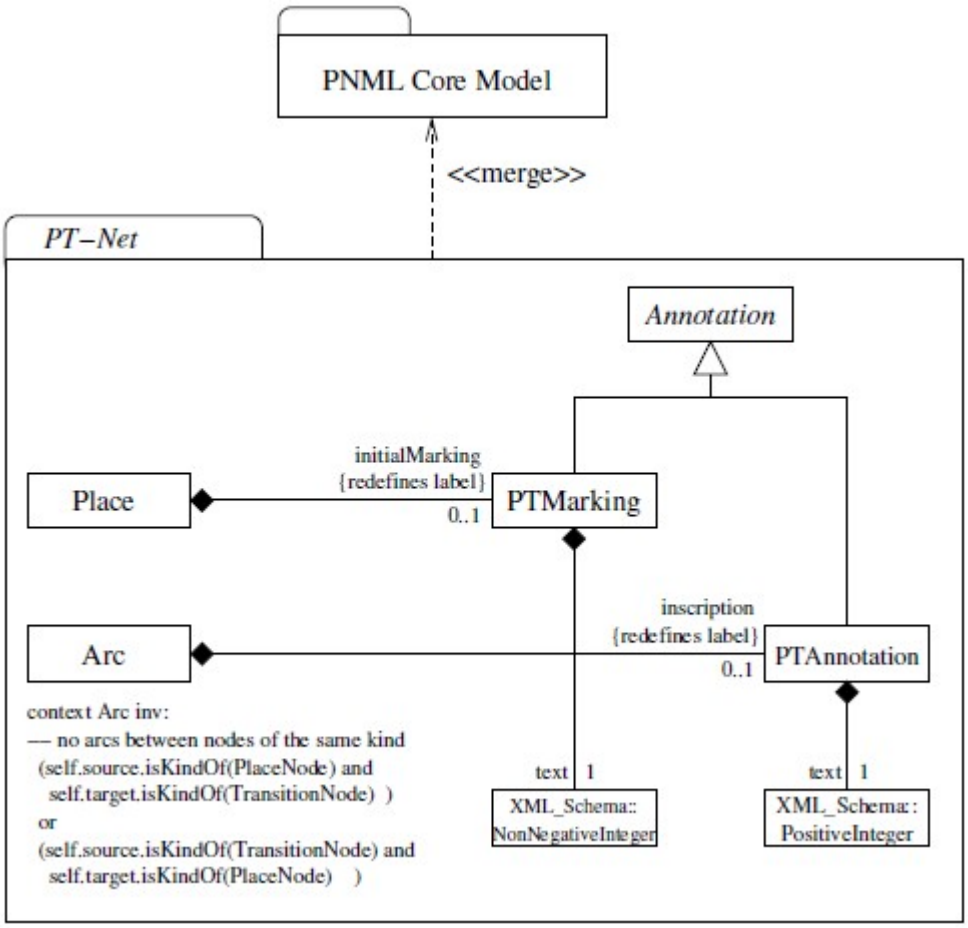



Figura 5: *Package P/T-Net* ([Kin06]).

Neste modelo, as únicas classes definidas são *PTMarking* e *PTAnnotation*. As restantes classes vêm do *package PNML Core Model*.

Cada *place* pode conter um *PTMarking* representado por um número natural. Em caso de omissão desta *label* é assumido que toma o valor de 0. Cada *arc* pode conter um *PTAnnotation* que consistem num número natural diferente de zero. Em caso de omissão desta *label* é assumido que tenha o valor de 1. O *PTMarking* é a *label* que contém informação relativa ao número inicial de marcas de um lugar, a *label PTAnnotation* tem informação do peso de um arco.

Nesta secção e na anterior foram apresentados os conceitos dos modelos *PNML Core Model* e *P/T-Nets*, a sintaxe XML que representa estes conceitos irá ser apresentada na secção seguinte.

### 3.4.3 Sintaxe PNML

Esta secção apresenta a sintaxe XML do *PNML Core Model* e do *Place/Transition-Net*, fazendo o mapeamento das classes concretas definidas nestes meta modelos a elementos XML.

As classes concretas são as que não estão representadas com o seu nome a itálico nos meta modelos.

Tabela 3.3: Mapeamento dos conceitos dos modelos *PNML Core Model* e *Place/Transition-Net* a elementos do PNML.

| Classe XML | Elemento XML | Attributos |
|---|---|---|
| PetriNetDoc | <pnml> | |
| PetriNet | <net> | id:ID<br>type:anyURL |
| Place | <place> | id:ID |
| Transition | <transition> | id:ID |
| Arc | <arc> | id:ID<br>source:IDRef (Node)<br>target:IDRef (Node) |
| Pag | <page> | id:ID |
| RefPlace | <referencePlace> | id:ID<br>ref:IDRef (Place or RefPlace) |
| RefTrans | <referenceTransition> | id:ID<br>ref:IDRef (Transition or RefTrans) |
| ToolInfo | <toolspecific> | tool:string<br>version:string |
| Name | <name> | |
| PTMarking | <initialMarking> | |
| PTAnnotation | <inscription> | |

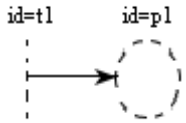
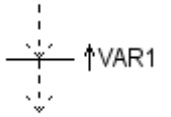
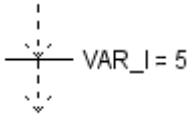
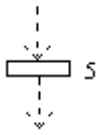
A Tabela 3.3 apresenta o mapeamento das classes do *PNML Core Model* e *Place/Transition-Net* a elementos XML juntamente com os seus atributos. O tipo de data ID corresponde a identificadores únicos no documento, o tipo de data IDRef refere-se a IDs que ocorrem no documento, ou seja, são referências para IDs.

Todos os elementos XML que correspondem a uma *label* do modelo *PNML Code Model* têm que conter o elemento `<text>` que representa o valor da *label*.

Quando uma rede de Petri modela o comportamento de um sistema, os seus lugares e transições estão relacionados com sinais de saída e entrada desse sistema, respectivamente. Desse modo, o tradutor considera que a descrição de um lugar corresponde a uma variável de saída booleana, que é verdadeira se o lugar possuir marcas, ou falsa no caso contrário. Por outro lado, considera que os nomes das transições correspondem a condições contendo variáveis de entrada, que podem ser do tipo booleano ou de outro tipo definido pelo utilizador. O tradutor permite o uso de transições temporizadas, que são uma extensão às redes de Petri em que uma transição só fica habilitada a partir de um tempo indicado, permite o uso de variáveis do tipo

flanco ascendente e descendente, e permite também o uso de variáveis dos tipos definidos em IEC 61131-3. A tabela seguinte apresenta estas extensões permitidas pelo tradutor e a representação das mesma em PNML.

Tabela 3.4: Sintaxe PNML dos elementos duma rede de Petri.

| Elemento | Nome/Descrição | Código PNML |
|---|---|---|
|  | **Lugar** | `<place id="p1">`<br>`  <name>`<br>`  <text>P3</text>`<br>`  </name>`<br>`</place>` |
|  | **Arco** | `<arc id="a1" source="p1" target="t1">`<br>`  <inscription>`<br>`  <text>2</text>`<br>`  </inscription>`<br>`</arc>` |
|  | **Detecção de flancos das variáveis**[1]: Os nomes das variáveis poderão ser antecedidos com “RE#” ou “FE#”para indicar que se trata de uma variável do tipo flanco ascendente ou descendente. | `<transition id="t1">`<br>`  <name>`<br>`  <text>RE#VAR1</text>`<br>`  </name>`<br>`</transition>` |
|  | **Variáveis não booleanas**[1]: Os nomes das variáveis poderão ser antecedidos pelo tipo de dados e por um “#” para indicar que a variável é desse tipo. No exemplo, a condição só dispara se a variável inteira VAR_I tiver o valor 5. | `<transition id="t1">`<br>`  <name>`<br>`  <text>INT#VAR_INT=5 </text>`<br>`  </name>`<br>`</transition>` |
|  | **Transição temporizada**[1]**:** Para indicar que uma transição é temporizada, o utilizador deverá inserir na etiqueta da transição “T#*s” em que o * deverá ser substituído por um número, que corresponde ao tempo em segundos. | `<transition id="t1">`<br>`  <name>`<br>`  <text>T#5s</text>`<br>`  </name>`<br>`</transition>` |

Um exemplo de um documento PNML completo pode ser encontrado no Anexo A.

1 As notações usadas para representar: uma transição temporizada, T#*s, o tipo de dados de uma variável, [tipo-de-dados]#[nome-variável], e os símbolos dos flancos das variáveis, RE#* e FE#*, não fazem parte do PNML. São notações utilizadas pela aplicação WEBGRAF 2 para conseguir interpretar esses símbolos

## 3.5 JARP

JARP [JARP01] é uma aplicação que permite a composição e análise de redes de Petri, guardando as redes num ficheiro com o formato PNML. Na altura em que JARP foi desenvolvido, o PNML ainda se encontrava em discussão e ainda não estava normalizado. E então, como o leitor já poderá ter adivinhado, nessa altura a sintaxe PNML era diferente da actual. Portanto, tiveram que ser feitas modificações à ferramenta JARP para que esta estivesse em conformidade com a versão actual do PNML. As modificações a esta ferramenta foram possíveis visto ela ser uma ferramenta opensource. Esta nova versão da ferramenta JARP pode ser encontrada em [Web10].

# 4 GRAFCET

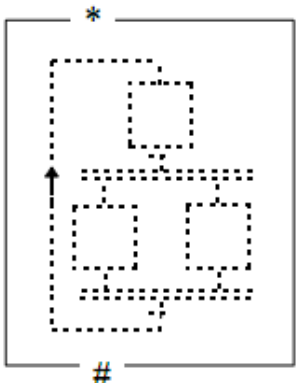
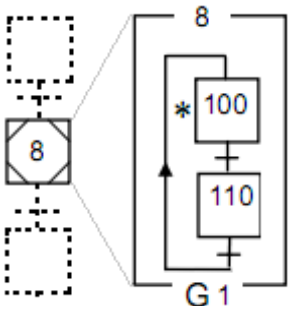
GRAFCET (GRAphe Fonctionnel de Commande Etape Transition) [IEC02] é uma linguagem de especificação para a descrição funcional do comportamento da parte sequencial de um sistema de controlo. Nesta secção irão ser apresentados os símbolos e as regras para a representação gráfica desta linguagem.

## 4.1 Items e regras básicas

A linguagem de especificação GRAFCET permite representar o comportamento de um sistema sequencial. Esta linguagem é caracterizada pelos seus elementos gráficos: as etapas, que poderão estar ou não activas; as transições, que indicam a possível evolução de uma etapa para outra; arcos orientados, que ligam transições a etapas e etapas a transições; e acções, responsáveis pela actualização das variáveis de saída. O conjunto de etapas activas em cada momento corresponde à situação actual do grafcet. A evolução de uma situação para outra depende das transições, onde cada uma é caracterizada pelas suas etapas de entrada, pelas suas etapas de saída e pela sua receptividade.

## 4.2 Representação gráfica dos elementos

Tabela 4.1: Representação gráfica dos lugares.

| Símbolo | Nome/Descrição | Exemplos | * |
|---|---|---|---|
|  | **Etapa:** Uma etapa é representada por um rectângulo. Num dado momento pode estar activa ou inactiva. O conjunto de etapas activas define a situação de um dado sistema. Uma etapa activa poderá ser marcada com um testemunho. Cada etapa tem uma etiqueta associada, para efeitos de identificação. A etiqueta, que está representada por um asterisco deverá conter um valor alfanumérico único em todo o GRAFCET. | Etapa 2:<br>  <br>Etapa 3 activa:<br>  | X |
| X* | **Variável de etapa (*step variable*):** A variável booleana X*, em que o asterisco corresponde à etiqueta da etapa, representa o estado da etapa *, tomando os valores lógicos “1” ou “0” dependendo se a etapa está activa ou inactiva. | Variável da etapa 10: X10. | X |
|  | **Etapa inicial:** Etapa que começa activa na situação inicial do sistema |  | X |
|  | **Etapa encapsuladora (*enclosing step*):** Este símbolo significa que a etapa * encapsula o grafcet parcial #. Quando esta etapa é activada, as etapas iniciais do grafcet parcial encapsulado (representadas com um asterisco à sua esquerda) são activadas, e a partir daí o grafcet parcial evolui naturalmente. A desactivação da etapa encapsuladora leva à desactivação de todas as etapas do grafcet parcial encapsulado por ela.<br>  | Enclosing step 8:<br>  <br>Quando a etapa 8 fica activa a etapa inicial 100 também é activada. A desactivação da etapa 8 leva à desactivação de todas as etapas do grafcet parcial G1. | |

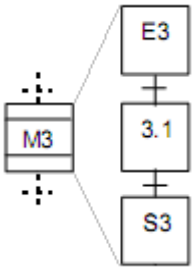
| | | | |
|---|---|---|---|
| X*/G# | **Designação global de uma encapsulação:** Um grafcet parcial # de uma etapa encapsuladora * pode ser descrito por uma expressão textual em que a etapa encapsuladora * é designada pela variável de etapa X*, a encapsulação pelo símbolo "/" e o grafcet parcial # por G#. | | |
| X*/X# | **Designação elementar de uma encapsulação:** Esta expressão textual indica que uma etapa # está encapsulada por uma etapa encapsuladora *. A expressão pode conter vários níveis de encapsulamento. | "X1/X5/X7" designa a etapa 7 encapsulada pela etapa 5 que por sua vez se encontra encapsulada pela etapa 1. | |
|  | **Etapa encapsuladora inicial:** Esta notação indica que esta etapa começa activa na situação inicial do sistema, o que implica que pelo menos uma das suas etapas encapsuladas será também uma etapa inicial. | | |
|  | **Macro-etapa:** Uma macro-etapa encapsula um grafcet em que E* é a etapa de entrada e S* é a etapa de saída. Quando M* é activo, E* fica activo também. As transições de saída de M* só poderão disparar quando a etapa S* estiver activa. O disparo da transição leva à desactivação da etapa S*. | Macro-Etapa M3:<br>  | |
| XM* | **Macro-etapa:** Uma macro-etapa está activa quando pelo menos um das suas etapas está activa. O estado de uma macro-etapa é representado pelos valores lógicos "1" ou "0" da variável XM*, em que o * é o nome da macro-etapa. | | |

Tabela 4.2: Representação gráfica das transições.

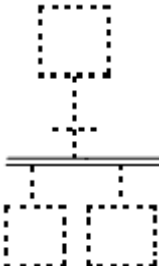
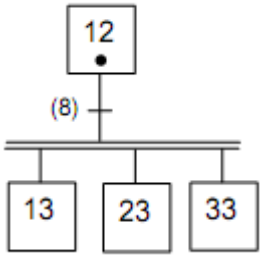
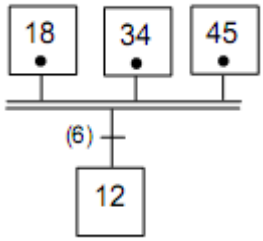
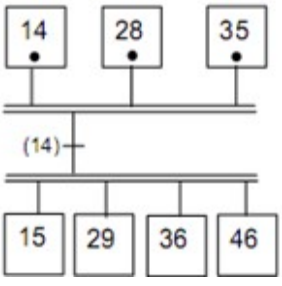
| Símbolo | Nome/Descrição | Exemplos | |
|---|---|---|---|
|  | **Transição:** Uma transição é representada por uma linha perpendicular aos arcos que a ligam às etapas de entrada e de saída. Só é permitido haver uma transição entre duas etapas. | | X |
|  | **Designação de uma transição:** As transições podem ter uma designação, geralmente colocada à sua esquerda. O asterisco poderá tomar qualquer valor alfanumérico. |  A transição (6) dispara quando a etapa 12 estiver activa | X |
|  | **Sincronização:** O símbolo da sincronização é representado por duas linhas horizontais paralelas. Este símbolo deve ser usado quando várias etapas ligam à mesma transição, agrupando os arcos.<br>No exemplo 1 a transição (8) dispara quando a etapa 12 está activa, o que resulta na activação das etapas 13, 23 e 33.<br>No exemplo 2 a transição (6) dispara quando todas as etapas antecedentes estão activas. A etapa 12 passa então a estar activa e as etapas 18, 34 e 45 são desactivadas.<br>No exemplo 3 a transição (14) dispara quando todas as etapas antecedentes estão activas (etapas 14, 28 e 35), o que resulta na activação das etapas 15, 29 36 e 46 e desactivação das etapas antecedentes | Exemplo 1:<br>  <br>Exemplo 2:<br>  <br>Exemplo 3:<br>  | X |

Tabela 4.3: Representação gráfica dos arcos orientados.

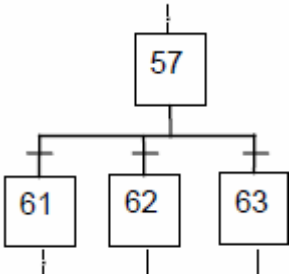
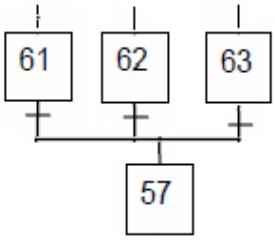
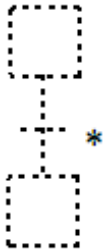
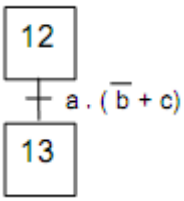
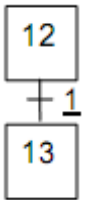
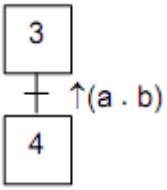
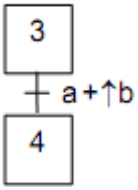
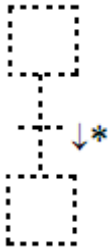
| Símbolo | Nome/Descrição | Exemplos | |
|---|---|---|---|
|  | **Arco orientado:** Os arcos orientados ligam etapas a transições e transições a etapas. Por convenção a orientação de um arco é de cima para baixo. Se a convenção não puder ser respeitada dever-se-á usar uma seta para indicar a orientação do arco, como ilustrado na figura do exemplo 1.<br>Quando uma etapa tem várias transições de saída, o arco que sai da etapa liga a um ponto de divergência que depois liga a cada uma das transições - exemplo 2. Se várias transições ligam a uma mesma etapa, então os arco que saem das transições deverão convergir num ponto que ligará à etapa - exemplo 3. | Exemplo 1:<br>  <br>Exemplo 2:<br>  <br>Exemplo 3:<br>  | X |

Tabela 4.4: Representação gráfica das receptividades.

| Símbolo | Nome/Descrição | Exemplos | |
|---|---|---|---|
|  | **Receptividade:** Cada transição tem associada uma proposição lógica chamada receptividade, que pode tomar os valores de "verdadeiro" e "falso". A proposição lógica compreende uma ou várias variáveis booleanas. A receptividade pode tomar o valor "1" para indicar que é sempre verdadeira. | Receptividade descrita por uma expressão booleana:<br>  <br><br>Receptividade sempre verdadeira:<br>  <br> | X |
|  | **Flanco ascendente de uma variável lógica:** O símbolo do flanco ascendente significa que a receptividade fica habilitada quando há uma mudança do seu valor de falso para verdadeiro. O flanco ascendente pode ser aplicado a toda a receptividade ou variáveis lógicas da expressão.<br>No exemplo 1 a receptividade é verdadeira quando o produto lógico "a . b" sofre uma mudança de valor de 0 para 1.<br>No exemplo 2 a receptividade é verdadeira quando "a" é verdadeiro e é detectada uma mudança de 0 para 1 da variável "b". | Exemplo 1:<br>  <br><br>Exemplo 2:<br>  <br> | X |
|  | **Flanco descendente de uma variável lógica:** O símbolo do flanco descendente significa que a receptividade fica habilitada quando há uma mudança do seu valor de verdadeiro para falso. O flanco descendente pode ser aplicado a toda a receptividade ou variáveis lógicas da expressão. | | X |

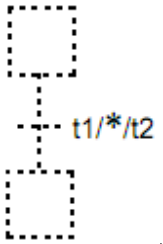
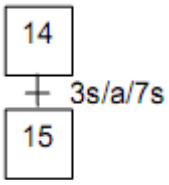
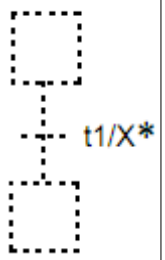
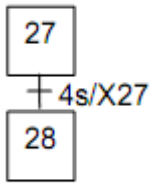
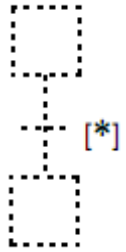
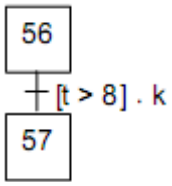
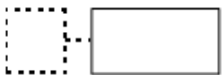
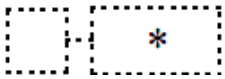
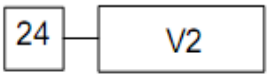
| | | | |
|---|---|---|---|
|  | **Receptividade temporizada:** A notação "t1/*/t2" indica que a receptividade é verdadeira t1 unidades de tempo após ter sido detectado um flanco ascendente de *. E volta a falso t2 unidades de tempo após acontecer um flanco descendente de *.<br>A receptividade da figura do exemplo é verdadeira 3 segundos após a mudança do sinal de "a" de 0 para 1. Torna-se falsa após 7 segundos da mudança do sinal de a de "1" para "0". |  | |
|  | **Receptividade temporizada (simplificação):** Nesta receptividade temporizada o asterisco deverá ser substituído por uma etiqueta duma etapa. A transição dispara t1 unidades de tempo após a activação da etapa *, e se ela se mantiver activa durante esse tempo.<br>A receptividade do exemplo fica habilitada se a etapa 27 se se mantiver 4 segundos activa. O que leva à activação da etapa 28 e desactivação da etapa 27. |  | X |
|  | **Asserção:** A expressão duma receptividade pode conter uma asserção dentro de parêntesis rectos. A expressão "[*]" é verdadeira se a asserção * se verificar, ou falsa caso contrário.<br>Na figura do exemplo, para a transição ser habilitada o valor da variável "8" tem que ser maior que 8 e "k" tem que ter valor lógico verdadeiro. |  | X |

Tabela 4.5: Representação gráfica das acções contínuas.

| Símbolo | Nome/Descrição | Exemplos | |
|---|---|---|---|
|  | **Acções contínuas:** Uma acção contínua está associada a uma etapa e é representada por um rectângulo. Várias acções podem estar associadas a uma mesma etapa. | | X |
|  | **Etiqueta de atribuição duma variável de saída:** Cada acção pode ter uma etiqueta com a designação duma variável de saída. A variável toma o valor verdadeiro enquanto a etapa, a que está associada a acção, estiver activa. |  | X |

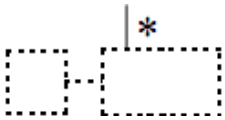
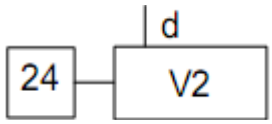
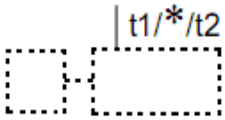
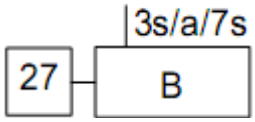
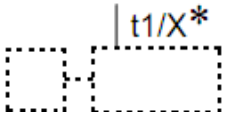
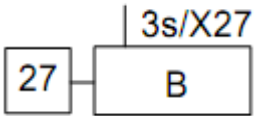
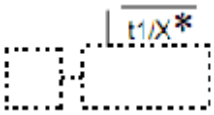
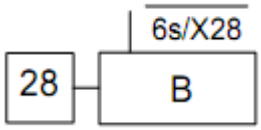
| | | | |
|---|---|---|---|
|  | **Condição de atribuição:** Uma acção contínua pode conter uma condição de atribuição, que é uma proposição lógica. Em caso de ausência desta notação a condição é sempre verdadeira. A condição de atribuição não deverá incluir flancos ascendentes ou descendentes das variáveis. A acção realizar-se-á se a etapa a que está associada estiver activa e se a condição de atribuição for verdadeira. |  A saída V2 toma o valor verdadeiro se a etapa 24 está activa e d é verdadeiro. Caso contrário toma o valor falso. Por outras palavras: V2 = X24 . d | X |
|  | **Condição de atribuição temporizada:** Esta notação indica que a condição de atribuição é verdadeira t1 unidades de tempo após a ocorrência de um flanco ascendente da variável *. E torna-se falsa t2 unidades de tempo após a ocorrência de um flanco descendente da variável *. |  | |
|  | **Acção com atraso:** Uma acção com atraso é uma acção contínua em que a condição de atribuição só é verdadeira t1 unidades de tempos após a activação da etapa *.<br>Na figura do exemplo, a saída B passa a verdadeiro 3 segundos após a activação da etapa 27. Se a etapa 27 voltar a ficar inactiva antes dos 3 segundos passarem, não chega a ser atribuído o valor verdadeiro a B. |  | X |
|  | **Acção de tempo limitado:** Esta é uma acção em que a condição de atribuição é verdadeira por um tempo igual a t1 após a etapa * ficar activa.<br>Na figura do exemplo, quando a etapa 28 fica activa é atribuído o valor verdadeiro à saída B por 6 segundos. |  | |

Tabela 4.6: Representação gráfica das acções de alocação.

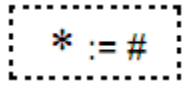
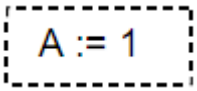
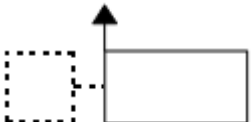
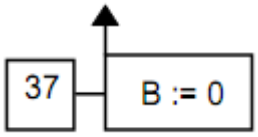
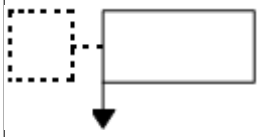
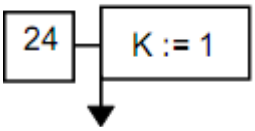
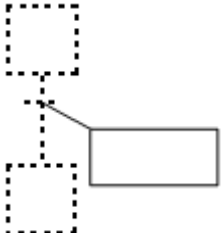
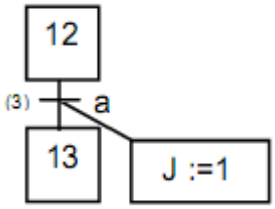
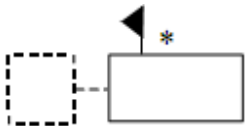
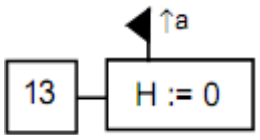
| Símbolo | Nome/Descrição | Exemplos | |
|---|---|---|---|
|  | **Alocação:** Uma alocação é definida textualmente por "* := #", em que é alocado o valor de # à variável * aquando da ocorrência de um evento interno associado às acções apresentadas a seguir nesta tabela. |  | X |
|  | **Acção na activação:** Esta acção está associada ao conjunto de eventos internos que leva à activação da etapa a que está associada. |  O valor 0 é alocado a B quando um evento leva à activação da etapa 37. | X |
|  | **Acção na desactivação:** Esta acção está associada ao conjunto de eventos internos que têm como consequência a desactivação da etapa a que está associada. |  A desactivação da etapa 24 leva à alocação do valor 1 a K. | X |
|  | **Acção na transposição:** Esta acção está associada ao conjunto de eventos internos que têm como consequência a transposição da transição a que está associada. |  A transposição da transição (3) | X |
|  | **Acção no evento:** Acção associada ao evento interno descrito pela expressão *. Por outras palavras: a ocorrência de * leva à execução da alocação definida na acção. Note-se que é condição a etapa a que está associada esteja activa. |  "H := 0" ocorre sempre que é detectado um flanco ascendente da variável "a" e simultaneamente a etapa 13 está activa. | X |

Tabela 4.7: Representação gráfica dos comentários.

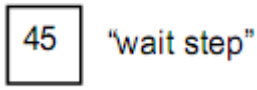
| **Símbolo** | **Nome/Descrição** | **Exemplos** | |
|---|---|---|---|
| "*" | **Comentário:** Os comentários relativos a elementos gráficos podem ser colocados entre aspas. |  Comentário "wait step" associado à etapa 45. | |

Tabela 4.8: Grafcets parciais.

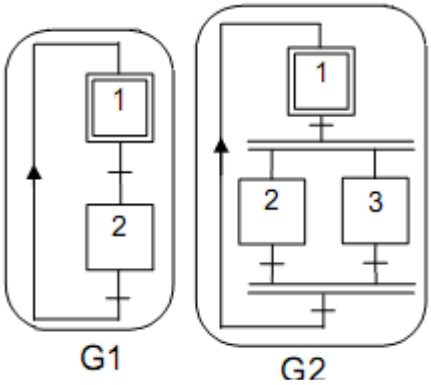
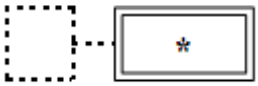
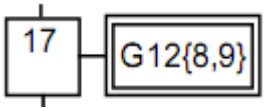
| **Símbolo** | **Nome/Descrição** | **Exemplos** | ***** |
|---|---|---|---|
| G* | **Nome de um grafcet parcial:** A letra G, por convenção indica um grafcet parcial, o * corresponde ao nome do grafcet parcial. A figura do exemplo demonstra um grafcet constituído por dois grafcets parciais G1 E G2. |  | X |
| XG* | **Variável de um grafcet parcial:** Um grafcet parcial está activo se pelo menos uma das suas etapas estiver activa. O estado de um grafcet parcial pode ser representado pelos valores lógicos "1" ou "0" da variável XG*. | | |
| G#{...,...} | **Situação de um grafcet parcial:** A situação de um grafcet parcial é representada pelo conjunto das suas etapas activas no momento considerado. | G12{1,3,8}: indica a situação do grafcet parcial 12 em que as suas etapas activas são a 1,3 e 8. | X |
| G#{*} | **Situação actual do grafcet parcial:** O asterisco indica a situação do grafcet parcial # no dado momento. O asterisco não deverá ser substituído. | | |
| G#{} | **Situação vazia do grafcet parcial:** Designa a situação do grafcet parcial # em que nenhuma etapa está activa. | | X |
| G#{INIT} | **Situação inicial do grafcet parcial:** Designa a situação inicial do grafcet parcial #. | | X |

Tabela 4.9: Representação gráfica da *forcing order*.

| Símbolo | Nome/Descrição | Exemplos | * |
|---|---|---|---|
|  | **Forcing order:** O asterisco deste símbolo deverá ser substituído por um dos símbolos da Tabela 4.8. Uma *forcing order* representa uma ordem interna que impõe uma situação num grafcet parcial hierarquicamente inferior. O grafcet forçado fica congelado e não pode evoluir durante o período da *forcing order*. O uso de *forcing orders* requer o uso de uma estrutura hierárquica com grafcets parciais de tal forma que a etapa, a que a *forcing order* está associada, faça parte de um grafcet de nível superior ao do grafcets parcial forçado. |  <br>Enquanto a etapa 17 estiver activa é imposta a situação G12{8,9}, (as únicas etapas activas do grafcet parcial 12 são a 8 e 9).<br>  <br>Enquanto a etapa 48 estiver activa, o garfcet parcial 3 vai-se manter na sua situação actual.<br>  <br>Enquanto a etapa 23 estiver ativa é imposta a situação vazia ao grafcet parcial 4. | X |

## 4.3 Explicitação textual para especificação de GRAFCETs

Actualmente, não existem tendências no sentido de normalizar uma linguagem textual para especificação de GRAFCETs. O mais próximo que existe é a linguagem PLCOpen XML, da organização PLCOpen [PLCO03], que contém a sintaxe precisa, em XML, para especificação da linguagem estrutural Senquential Function Charts (SFC), entre outras. Apesar de GRAFCET e SFC serem duas linguagens diferentes, tanto em termos de utilização como de sintaxe (GRAFCET é uma linguagem de modelação, SFC é uma linguagem de programação), elas possuem alguns elementos em comum (visto o SFC ser baseado em GRAFCET), e foi essa a razão que levou à escolha da linguagem PLCOpen XML como ponto de partida para a criação de um formato textual de GRAFCET. Esta linguagem tem como nome Grafcet Markup Language (GML).

De seguida é apresentado, na Tabela 4.10, o mapeamento dos elementos constituintes de um grafcet a elementos XML do GML.

Tabela 4.10: Mapeamento dos elementos constituintes de um grafcet para elementos XML do GML.

| Elemento GRAFCET | Elemento XML | Attributos |
|---|---|---|
| Grafcet | <gml> | |
| Grafet parcial | <partialGrafcet> | localId:ID<br>name:string |
| Etapa | <step> | name:string<br>localId:ID<br>initialStep:boolean |
| Transição | <transition> | localId:ID |
| Acção | <actionBlock> | |
| Forcing order | <forcingOrder> | |

Um documento GML tem como elemento raiz `<gml>`. No caso de existirem grafcets parciais, o elemento raiz é constituído pelos elementos XML que os representam - `<partialGrafcet>`. O elemento `<partialGrafcet>` contém as etapas que constituem o grafcet parcial respectivo. Note-se que o elemento `<partialGrafcet>` poderá também conter as transições e acções ligadas às etapas, mas, por definição, só é necessário conter as etapas. As transições e acções poderão ficar contidas directamente no elemento `<gml>`. No caso de não existirem grafcets parciais, o elemento raiz possui os elementos das etapas - `<step>`.

A Tabela 4.10 não inclui o elemento *sincronização.* Isto deve-se ao PLCOpen XML diferenciar sincronizações de divergência e convergência. Assim, há dois elementos XML para representar uma *sincronização*. Eles estão representados na Tabela 4.11. Estão ainda também representados outros elementos que correspondem aos pontos de convergência e divergência quando uma transição liga a várias etapas ou vice-versa (exemplos 2 e 3 da Tabela 4.3).

Tabela 4.11: Mapeamento dos elementos de divergência e convergência de arcos, do tipo “e” e “ou”, a elementos XML.

| Elemento GRAFCET | Elemento XML | Attributos |
|---|---|---|
| Sincronização divergente | <simultaneousDivergence> | localId |
| Sincronização convergente | <simultaneousConvergence> | localId |
| Divergência de arcos | <selectionDivergence> | localId |
| Convergência de arcos | <selectionConvergence> | localId |

A estrutura básica de todos os elemento GML está a seguir apresentada:

<elemento>

<posição_do_elemento>

<referências_para_outros_elementos>

Cada elemento inclui informação sobre a sua posição absoluta e referências para outros elementos a que esteja ligado. Cada elemento poderá ainda conter outros elementos que sejam próprios dele. Por exemplo: a condição de uma transição. A posição é dada pelo elemento XML `<position>` que possui os atributos `x` e `y` do tipo inteiro, que correspondem às coordenadas cartesianas do respectivo elemento. O elemento XML `<connectionPointIn>` é constituído por um ou vários elementos XML `<connection>`. Este elemento tem como atributo `refLocalId` que representa uma referência para um *id* no documento GML.

Os símbolos usados para representar flancos ascendentes ("↑") e descendentes ("↓") deverão, logicamente, ser representados por uma combinação de caracteres ASCII no ficheiro que contém a explicitação textual do modelo dum sistema. A tabela seguinte faz a correspondência de cada um destes símbolos com a representação textual admitida pelo tradutor.

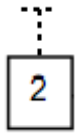
Tabela 4.12: Representação textual dos flancos.

| Tipo de flanco | Representação textual |
|---|---|
| ↑ | RE# |
| ↓ | FE# |

Assim, uma transição que tenha como condição "↑var1 AND ↓var2", textualmente, deverá ser representada da seguinte maneira: "RE#var1 AND FE#var2", de forma a poder ser interpretada pelo tradutor.

As sub-secções seguintes irão apresentar a estrutura completa da sintaxe dos elementos do GRAFCET.

### 4.3.1 Etapa

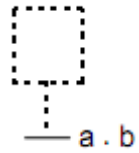



```
<step name="2" localId="1">
    <position x="10" y="10" />
    <connectionPointIn>
        <connection refLocalId="3" />
    </connectionPointIn>
</step>
```

Figura 4.1: Representação em GML de uma etapa.

O elemento `<step>` tem como argumentos `name`, `localId` e `initialStep`. A omissão do atributo `initialStep` corresponde a ter o valor *false*. Uma etapa contém a estrutura básica, que foi anteriormente apresentada, em que o atributo `refLocalId` é uma referência para a transição de entrada da etapa.

### 4.3.2 Transição

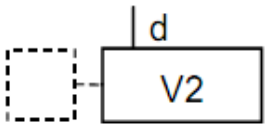



```
<transition localId="3">
     <position x="20" y="16" />
     <connectionPointIn>
         <connection refLocalId="1" />
     </connectionPointIn>
     <condition>
          <reference name="a.b" />
     </condition>
</transition>
```

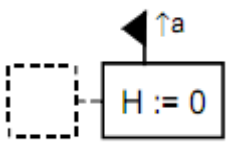
Figura 4.2: Representação em GML de uma transição.

O elemento `<transition>` contém o atributo `localId`. O sub-elemento `<condition>` contém a condição da transição no atributo `name` do seu sub-elemento `<reference>`. O atributo `refLocalId` refere-se ao *id* da etapa de entrada.

### 4.3.3 Acção




```
<actionBlock>
     <position x="20" y="30" />
     <connectionPointIn>
          <connection refLocalId="14" />
     </connectionPointIn>
     <action qualifier="continuous">
          <reference name="V2" />
          <condition name="d" />
     </action>
</actionBlock>
```




```
<actionBlock>
     <position x="10" y="10" />
     <connectionPointIn>
          <connection refLocalId="15" />
     </connectionPointIn>
     <action qualifier="onEvent">
          <reference name="H := 0" />
```

```
                    <condition name="RE#a" />
               </action>
          </actionBlock>
```

Figura 4.3: Representação em GML de acções.

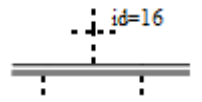
Uma acção é representada pelo elemento `<actionBlock>`. Este contém o sub-elemento `<action>` que é constituído pelos elementos `<reference>` e `<condition>`. O atributo `qualifier` pode conter um dos seguintes valores:

- continuous
- onEvent
- onActivation
- onDeactivation
- clearing

`<reference>` tem como atributo: `name`, que representa a variável de saída, no caso de ser uma acção contínua, ou uma alocação (exemplo: A := 1) no caso de ser uma acção de activação, desactivação, evento ou na transposição. `<condition>` tem o atributo `name` que representa a condição da acção no caso de ser acção continua ou de evento. A ausência deste elemento indica que a condição é sempre verdadeira *(true)*. No caso das acções contínuas ou de eventos, os símbolos para representar flancos ascendentes ("↑") ou descendentes ("↓") deverão ser substituídos por "RE#" e "FE#", respectivamente.

### 4.3.4 Sincronização


```
<simultaneousConvergence localId="19">
     <connectionPointIn>
          <connection refLocalId="12" />
          <connection refLocalId="14" />
     </connectionPointIn>
</simultaneousConvergence>
```

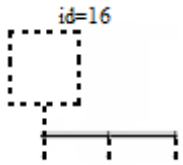



```
<simultaneousDivergence localId="20">
     <connectionPointIn>
          <connection refLocalId="16" />
     </connectionPointIn>
</simultaneousDivergence>
```

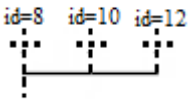
Figura 4.4: Representação em GML de sincronizações.

No caso do elemento `<simultaneousConvergence>`, os atributos `refLocalId` são referências para *ids* das etapas de entrada. No caso do elemento `<simultaneousDivergence>` só existe um elemento `<connection>` e o seu atributo `refLocalId` é uma referência que corresponde ao *id* duma transição.

### 4.3.5 Convergência e divergência de arcos




```
<selectionDivergence localId="20">
     <connectionPointIn>
          <connection refLocalId="6" />
     </connectionPointIn>
</selectionDivergence>
```

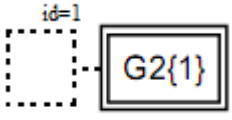



```
<selectionConvergence localId="22">
     <connectionPointIn>
          <connection refLocalId="8" />
          <connection refLocalId="10" />
          <connection refLocalId="12" />
     </connectionPointIn>
</selectionConvergence>
```

Figura 4.5: Represetação em GML de convergências e divergências de arcos.

`<selectionDivergence>` e `<selectionConvergence>` são os elementos XML que representam os pontos de divergência e convergência de arcos. `<selectionDivergence>` apenas contém um elemento `<connection>` cujo atributo `refLocalId` é uma referência para a etapa antecedente. Já o elemento `<selectionConvergence>` pode conter vários que representam os *ids* das transições precedentes.

### 4.3.6 Forcing Order

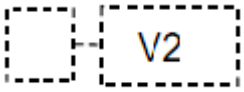



```
<forcingOrder>
     <connectionPointIn>
          <connection refLocalId="1" />
     </connectionPointIn>
     <situation>
          <reference name="G2{1}" />
     </situation>
</forcingOrder>
```

Figura 4.6: Representação em GML de uma *forcing order*.

Uma *Forcing Order* é representada pelo elemento `<forcingOrder>`. Como já anteriormente mencionado, o atributo `refLocalId` é uma referência para um *id* de um elemento XML contido no documento. No caso concreto das *Forcing Orders*, o *id* é relativo à etapa a que a *Forcing Order* está associada. O elemento `<situation>` representa a situação de um grafcet que deve ser forçada. A situação está contida no atributo `name` do elemento `<reference>`.

### 4.3.7 Variáveis




```
<localVars name="Local variables">
     <variable name="V2">
          <type>
               <BOOL />
          </type>
          <initialValue>
               <simpleValue value="FALSE" />
          </initialValue>
     </variable>
     ...
</localVars>
```

Figura 4.7: Represetação em GML de das variáveis do sistema.

O elemento `<localVars>` contém todas as variáveis usadas no documento GML. Cada variável é representada pelo element `<variable>` que tem o nome como atributo – `name`. O tipo da variável é dado pelo sub-elemento do elemento `<type>`, que pode ser qualquer um da seguinte lista:

- `<BOOL>`
- `<INT>`
- `<REAL>`

O valor inicial de cada variável é dado pelo atributo `<value>` do elemento `<simpleValue>`.

Um exemplo de um documento GML completo pode ser encontrado no Anexo B.

# 5 PLCs e Norma IEC 61131-3

Um PLC (Controlador Lógico Programável) é um dispositivo físico que desempenha funções de controlo. A sua arquitectura torna-o indicado para lidar com sistemas de eventos discretos (processos em que as variáveis se alteram bruscamente e assumem um conjunto finito de valores).

A Figura 5.1 ilustra a aplicação geral dum PLC.

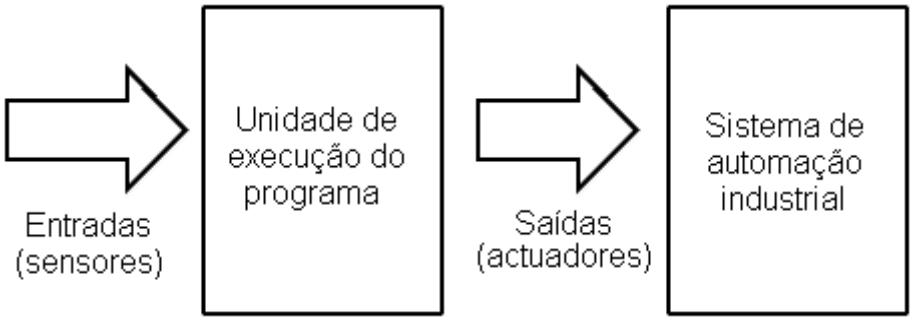



Figura 5.1: Aplicação geral dum PLC.

Os dispositivos de entrada lêem o estado do sistema. As entradas são processadas pelo PLC levando depois os dispositivos de saída a actuar sobre o sistema automatizado.

O documento IEC 61131-3 [IEC99] é a parte 3 da norma IEC 61131 e nele está especificada a sintaxe e semântica das linguagens de programação dos controladores lógicos programáveis. São elas: IL(Instruction List), ST (Structured Text), LD (Ladder Diagram), FBD (Function Block Diagram) e SFC (Sequential Function Chart).

Os capítulos seguintes irão apresentar a arquitectura e funcionamento geral dos PLCs. Seguidamente, irá ser feita uma breve apresentação de cada uma das linguagens textuais (IL e ST), começando por mencionar os elementos comuns a ambas.

## 5.1 Arquitectura geral de um PLC

Esta secção apresenta a estrutura básica de um PLC. Os elementos básicos que o constituem são a unidade de execução do programa, os módulos de Entrada/Saída e a fonte de alimentação, que vão ser detalhadas nas sub-secções seguintes.

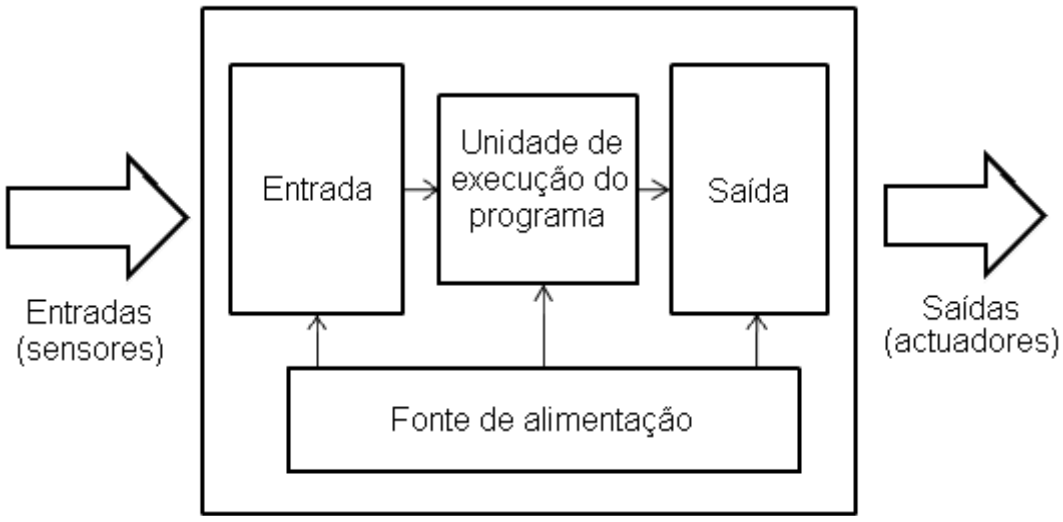



Figura 5.2: Arquitectura geral dum PLC.

### 5.1.1 Unidade de execução do programa

A unidade de execução do programa é o elemento central do PLC. É ele que armazena o programa de controlo do programador, e é o responsável pela leitura das entradas e afectação das saídas, bem como pela realização de operações matemáticas e lógicas internas. Toda a informação vinda das entradas é processada pela unidade de execução do programa, e os resultados são enviados para as saídas de acordo com as instruções do programa.

### 5.1.2 Módulos de Entrada/Saída

Os módulos de Entrada/Saída são responsáveis pela comunicação entre o CPU e os dispositivos que interagem com o ambiente - sensores e actuadores. As entradas recebem os sinais dos sensores e transformam esses sinais de forma a terem níveis adequados para poderem ser processados pelo CPU. As saídas enviam os sinais, obtidos pelo CPU, aos dispositivos externos para que estes actuem em conformidade.

### 5.1.3 Fonte de alimentação

A fonte de alimentação é o aparelho que fornece as tensões necessárias ao funcionamento de todos os componentes dum PLC.

## 5.2 Funcionamento geral de um PLC

Os PLCs executam ciclicamente os programas de controlo que os seus programadores neles descarregam. Cada ciclo é constituído por três fases:

**Fase 1: Leitura das entrada** – Na primeira fase (os sensores lêem grandezas físicas do ambiente e convertem-nas em sinais eléctricos passíveis de serem interpretados pelo PLC) a informação disponibilizada pelos sensores é guardada nas variáveis de entrada. Pode-se dar um exemplo de um termómetro que lê a temperatura duma sala e guarda a temperatura na variável de entrada adequada.

**Fase 2: Execução do programa de controlo** – Na segunda fase, o programa de controlo é executado tendo em conta as variáveis de entrada. Daí resulta a atribuição de valores resultantes do processamento das variáveis de entrada às variáveis de saída.

**Fase 3 Actualização das saídas** – Na terceira fase, que corresponde ao final do ciclo, os valores das variáveis de saída são transferidos para as saídas. Estas vão accionar os actuadores de acordo com o resultado do programa.

## 5.3 Elementos comuns às linguagens IEC 61131-3

Esta secção define os elementos comuns às linguagens de programação dos controlos programáveis definidas especificadas na Parte 3 da norma IEC 61131.

### 5.3.1 Tipos de dados

A norma define um conjunto de tipos de dados primitivos: BOOL, BYTE, WORD, DWORD, SINT, INT, UINT, REAL, TIME, DATE e STRING. Contudo, também permite que sejam definidos tipos de dados próprios, chamado de tipo derivado de dados. Este pode ser um tipo único (TYPE) ou uma composição de várias variáveis e tipos (STRUCT). O exemplo a seguir, apresenta a declaração de uma STRUCT:

Listagem 1: Exemplo de declaração de uma STRUCT.

```
1. TYPE
2.       Rectangulo :
3.        STRUCT
4.             Altura : UINT;
5.             Largura: UINT;
6.       END_STRUCT ;
7. END_TYPE
```

A STRUCT `Rectangulo`, possui dois atributos: `Altura` e `Largura`, ambos do tipo UINT (inteiro não negativo).

### 5.3.2 Variáveis

Variáveis são espaços reservados na memória do computador que guardam informação a ser utilizada durante o código do programa. As variáveis podem ser de dois tipos: locais e globais. As variáveis locais só podem ser usadas na unidade de organização de código onde foram declaradas. As variáveis globais podem ser usadas em qualquer unidade de organização de código, devendo ser declaradas no espaço reservado às variáveis globais (VAR_GLOBAL). A seguir, apresenta-se a declaração de uma variável de nome VariavelExemplo cujo tipo de dados é inteiro:

```
VariavelExemplo : INT;
```

Na declaração de uma variável, o seu nome deverá ser seguido do símbolo ":"(dois pontos) e do tipo de dados pretendido.

A cada variável pode ser-lhe atribuída um valor inicial, no momento em que é declarada. A seguir, apresenta-se a inicialização da variável anterior com o valor 5:

```
VariavelExemplo : INT := 5;
```

A declaração da variável deverá ser seguida de ":=" (dois pontos e igual) e do valor com que se pretende inicializar a variável.

A inicialização de uma variável do tipo `Rectangulo` (STRUCT definida na sub-secção anterior - 5.3.1) é feita da seguinte maneira:

```
variavelStruct : Rectangulo := (Altura:=5, Largura:=10);
```

Também é permitido o uso de *arrays*, o que permite que sejam guardados vários valores de um tipo na mesma variável. Um exemplo de uma declaração de um *array* é a seguir apresentado:

```
ArrayExemplo : ARRAY [1..5] OF INT;
```

O exemplo mostra a declaração de um *array* de 5 posições, a começar no índice 1, do tipo inteiro. Na declaração dum *array,* coloca-se o nome do *array,* seguido do símbolo ":" (dois pontos), da *keyword* ARRAY, do seu tamanho (colocando o primeiro e o último índice do array), da *keyword* OF e finalmente do tipo de dados. Para se aceder a um dos seus valores, coloca-se o índice da posição pretendida entre parêntesis rectos ("[", "]") à frente do nome do *array*:

```
ArrayExemplo[1] := 15;
```

O exemplo anterior corresponde a uma atribuição do valor 15 à posição 1 do *array,* na linguagem ST.

Um *array* poderia ser do tipo `Rectangulo`:

```
ArrayRectangulos : ARRAY [1..5] OF Rectangulo;
ArrayRectangulos[5] := (Altura := 2, Largura := 6);
```

E uma STRUCT pode conter *arrays* como atributos:

Listagem 2: Exemplo de declaração de uma STRUCT.

```
1. TYPE
2.       structExemplo :
3.        STRUCT
4.             atributo1 : ARRAY [1..3] OF INT;
5.             atributo2 : INT;
6.       END_STRUCT ;
7. END_TYPE
```

Para se aceder a um atributo de uma variável STRUCT, coloca-se o nome da variável seguido de um ponto (“.”) e do nome do atributo pretendido:

```
structExemplo.atributo1[1] := 5;
structExemplo.atributo2 := 5;
```

Os exemplos acima correspondem à atribuição do valor 5 à posição 1 do *array* `atributo1`, e ao atributo `atributo2`, em ST.

### 5.3.3 Unidades de Organização de Programas (Program Organization Units – POU)

A norma IEC 61131-3 define 3 unidades de organização de programas. São elas: Function, Function Block e Program. Cada uma delas tem uma zona de declaração de variáveis, e o corpo da unidade propriamente dito.

#### Funções (Function)

As Funções são pedaços de código que recebem parâmetros e retornam um valor que é o resultado da função. As funções são caracterizadas por retornarem sempre o mesmo valor para as mesmas variáveis de entrada. A Listagem 3 apresenta um exemplo de uma Function:

Listagem 3: Exemplo de uma Function em ST.

```
1. FUNCTION SIMPLE_FUN : REAL
2.       VAR_INPUT
3.             A, B : REAL;
4.             C : REAL := 1.0;
5.       END_VAR
6. SIMPLE_FUN := A*B/C;
7. END FUNCTION
```

Esta função, de nome SIMPLE_FUN, recebe como parâmetros A, B e C, em que C por omissão tem o valor de 1.0, e retorna o resultado da expressão A*B/C.

#### Blocos de funções (Function Block)

Uma Function Block é uma unidade de organização de programa que, quando executado, produz um ou mais valores. Podem ser criadas várias instâncias. Cada instância deve ter um identificador associado (o nome), e uma estrutura de dados contendo as suas variáveis de entrada, variáveis internas e variáveis de saída  Todos os valores das variáveis de saída e as variáveis internas necessárias desta estrutura de dados devem persistir de uma execução da Function Block para a próxima. Portanto, a invocação de uma Function Block com os mesmos argumentos (variáveis de entrada) não implica que produza sempre os mesmos valores de saída, ao contrário das Functions.

Listagem 4: Exemplo de uma Function Block em ST.

```
1.  FUNCTION_BLOCK RE_TRIG
2.        VAR_INPUT
3.              CLK: BOOL;
4.        END_VAR
5.        VAR_OUTPUT
6.              Q: BOOL;
7.        END_VAR
8.        VAR
9.              M: BOOL := FALSE;
10.       END_VAR
11.       Q := CLK AND NOT M;
12.       M := CLK;
13. END_FUNCTION_BLOCK
```

A Function Block apresentada na Listagem 4 consiste em detectar um Rising Edge de uma variável, ou seja, uma mudança de nível lógico de 0 para 1. Como variável de entrada tem a variável CLK do tipo booleano; como variável de saída tem a variável Q, também do tipo booleano. Apenas estas duas variáveis são visíveis fora da instância da Function Block. A variável M (booleana) é uma variável interna e guardada dentro da instância.

### Programas (Program)

Um Program consiste numa rede de Functions e Function Blocks, que podem trocar dados.

## 5.3.4 Variáveis globais (VAR_GLOBAL)

Quando há necessidade de variáveis globais, (e, como tal, acessíveis por qualquer POU no projecto) estas podem ser declaradas no grupo VAR_GLOBAL. Variáveis com grande necessidade de serem declaradas globalmente são variáveis ligadas a pontos de Entrada/Saída.

Listagem 5: Exemplo de VAR_GLOBAL.

```
1. VAR_GLOBAL
2.       variavel1 : INT;
3.       variavel2 : INT;
4.       array_de_ints : ARRAY [1.5] OF INT;
5. END_VAR
```

## 5.4 Linguagens de programação textuais

### 5.4.1 Instruction List (IL)

IL é uma linguagem de baixo nível semelhante a *Assembly*. É uma linguagem popular para algoritmos relativamente simples. Apesar de ser considerada por muitos a mais aborrecida de programar, é a mais rápida a executar. Consiste numa sequencia de instruções que são executadas pela ordem por que se encontram. Cada instrução deve começar numa nova linha e deve conter um operador com um ou mais operandos (separados por vírgulas). Esta linguagem possui um acumulador, que guarda um valor a ser usado pelos operadores. A Tabela 5.1 apresenta uma lista de operadores desta linguagem com a respectiva descrição e exemplo de utilização.

Tabela 5.1: Operadores da linguagem IL.

| **Operador** | **Descrição** | **Exemplo** |
|---|---|---|
| LD | Load: Carrega o valor do operando no acumulador. | LD 5<br>Acumulador fica com valor 5. |
| ST | Store: Armazena o valor do acumulator para a variável do operando. | LD 5<br>ST Var<br>Var fica com o valor 5 |
| AND | Realiza a operação AND com o acumulator e com o operando. | LD true<br>AND false<br>Acumulador fica com valor *false*. |
| ANDN | Realiza a operação AND com o acumulator e com a negação do operando. | LD true<br>ANDN false<br>Acumulador fica com valor *true*. |
| OR | Realiza a operação OR com o acumulator e com o operando. | LD false<br>OR true |

| | | |
|---|---|---|
| | | Acumulador fica com valor *true*. |
| ORN | Realiza a operação OR com o acumulator e com a negação do operando. | LD false<br>ORN true<br>Acumulador fica com valor *false*. |
| NOT | O valor actual do acumulador é negado. | LD true<br>NOT<br>ST Var<br>Var fica com valor *false.* |
| ADD | Adição do acumulador e do operando. O resultado é copiado para o acumulador | LD 6<br>ADD 2<br>ST Var<br>Var fica com o valor 8. |
| SUB | Subtracção do acumulator e do operando. O resultado é copiado para o acumulador. | LD 6<br>SUB 2<br>ST Var<br>Var fica com o valor 4. |
| MUL | Multiplicação do acumulator e do operando. O resultado é copiado para o acumulador. | LD 6<br>MUL 2<br>ST Var<br>Var fica com o valor 12. |
| DIV | Divisão do acumulator pelo operando. O resultado é copiado para o acumulador. | LD 6<br>DIV 2<br>ST Var<br>Var fica com o valor 3. |
| GT | Greater than (>): Verifica se o acumulador é maior que o operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false.* | LD 2<br>GT 1<br>Acumulador fica com valor *true.* |
| GE | Greater of equal (>=): Verifica se o acumulador é maior ou igual do que o operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false.* | LD 2<br>GE 2<br>Acumulador fica com valor *true.* |
| EQ | Equal (=): Verifica se o acumulador é igual ao operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false.* | LD 2<br>EQ 2<br>Acumulador fica com valor |

| | | |
|---|---|---|
| | | *true.* |
| NE | Not equal (<>): Verifica se o acumulador é diferente do operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false.* | LD 2<br>NE 2<br>Acumulador fica com valor false. |
| LE | Less or equal (<=): Verifica se o acumulador é menor ou igual ao operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false;* | LD 1<br>LE 2<br>Acumulador fica com valor *true.* |
| LT | Less than(<): Verifica se o acumulador é menor do que o operando. Em caso afirmativo, o acumulador fica com o valor *true*, caso contrário *false;* | LD 1<br>LE 2<br>Acumulador fica com valor *true.* |
| JMP | Salto incondicional para uma etiqueta | JMP FLAG |
| JMPC | Salto condicional para uma etiqueta. O salto é concretizado se o valor do acumulador for *true.* | LD Var<br>EQ 5<br>JMPC FLAG<br>O salto para a eiqueta FLAG é concretizado se a Var possuir o valor 5. |
| JMPCN | Salto condicional para uma etiqueta, after a check if the accumuluator is FALSE | LD Var<br>EQ 5<br>JMPCN FLAG<br>O salto para a eiqueta FLAG é concretizado se o valor de Var for diferente de 5. |
| CAL | Faz a chamada a uma *Function Block.* | CALL F_TRIG(Var); |
| RET | Retorno incondicional do POU e regresso ao POU que o chamou. | |
| RETC | Retorno condicional do POU e regresso ao POU que o chamou, se o valor do acumulador for *true*. | |
| RETCN | Retorno condicional do POU e regresso ao POU que o chamou, se o valor do acumulador for *false*. | |

### 5.4.2 Structured Text (ST)

ST é uma linguagem de alto nível que se assemelha a PASCAL. É uma linguagem de grande flexibilidade para algoritmos de controlo.

A atribuição de um valor a uma variável é feito colocando ":=" (dois pontos e igual) e o valor de atribuição a seguir ao nome da variável a que se pretende atribuir o valor:

```
VariavelExemplo := 5;
```

Neste exemplo, é atribuído o valor 5 a `VariavelExemplo`. A tabela que se segue, apresenta a lista dos operador desta linguagem.

Tabela 5.2: Operadores da linguagem ST.

| Operação | Símbolo | Exemplo |
|---|---|---|
| Negação | -, NOT | Var1 := -5;<br>Var2 := NOT Var3; |
| Exponenciação | ** | Var := 2**2;<br>Var fica com o valor 4 (o mesmo que 2 ao quadrado). |
| Multiplicação | * | ResultadoMultiplicacao := a * b; |
| Divisão | / | ResultadoDivisao := a / b; |
| Adição | + | ResultadoAdicao := a + b; |
| Subtracção | - | ResultadoSubtraccao := a – b; |
| Comparação | <, >, <=, >= | ResultadoComparação := Var > 3;<br>É atribuido o valor *true* à variável ResultadoComparação se Var for maior que 3, ou *false* em caso contrário. |
| Igualdade | = | ResultadoIgualdade := Var = 3;<br>É atribuido o valor *true* à variável ResultadoIgualdade se Var for igual a 3, ou *false* se for diferente. |
| Desigualdade | <> | ResultadoDesigualdade := Var <> 3;<br>É atribuido o valor *true* à variável ResultadoDesigualdade se Var for diferente de 3, ou *false* em caso contrário. |
| Operador 'e' | &, AND | Var = a AND b; |
| Operador 'ou' exclusivo | XOR | Var = a XOR b; |
| Operador 'ou" | OR | Var = a OR b; |

A tabela está ordenada por ordem decrescente de precedência. No entanto, podem ser usados parêntesis para alterar a sua ordem de precedência:

```
VarResultado := (Var1 + Var2) * Var3;

VarResultado2 := Var4 = (Var5 AND Var6);
```

No primeiro exemplo, o uso dos parêntesis faz com que a soma tenha precedência sobre a multiplicação. E no segundo exemplo, o operador 'e' (AND) tem precedência sobre o operador de igualdade ('=') devido aos parêntesis.

As instruções da linguagem ST estão sumarizadas na Tabela 5.3

Tabela 5.3: Instruções da linguagem ST.

| Tipo de instrução | Exemplos |
|---|---|
| IF | `IF a < b THEN`<br>`  c := 0;`<br>`ELSE`<br>`  c := 1;`<br>`END_IF` |
| CASE | `CASE var OF`<br>`  1:  c := 1;`<br>`  2:  c := 2;`<br>`  3:  c := 3;`<br>`ELSE`<br>`  c := 4;`<br>`END_CASE` |
| FOR | `FOR i := 1 TO 10 DO`<br>`  a[i] := 0;`<br>`END_FOR` |
| WHILE | `WHILE var < 10 DO`<br>`  var := var + 1;`<br>`END_WHILE` |
| REPEAT | `REPEAT`<br>`  var := var + 1;`<br>`UNTIL var = 10`<br>`END_REPEAT` |

## 5.5 Desenvolvimento e execução de um programa PLC

Antes de se passar para o Capítulo 6, onde é apresentada a arquitectura da aplicação e do código gerado por ela, é importante perceber como é executado um programa PLC e como é que se desenvolve software para PLCs que execute o programa. Analise-se para isso a seguinte figura:

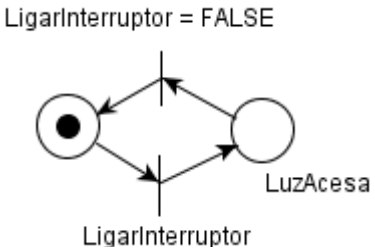



Figura 5.3: Rede de Petri para exemplificação do funcionamento de um programa PLC

A Figura 5.3 representa uma simples rede de Petri que simula o comportamento de um interruptor. Enquanto o interruptor permanecer ligado, uma luz acende-se. Quando o interruptor é desligado a luz apaga-se. O programa PLC, deverá executar de tal forma que, quando a variável de entrada LigarInterruptor ficar verdadeira, a variável de saída LuzAcesa fica também verdadeira. E quando LigarInterruptor passar a  falso, a variável LuzAcesa deverá também ficar falsa.

Tipicamente, numa simples máquina de estados (em que cada estado ou está activo ou desactivo, e apenas um estado está activo de cada vez), cria-se uma variável para indicar o estado actual do sistema, e encadeia-se um conjunto de condições IF-THEN-ELSE que tratam da evolução da máquina de estados. No fim, actualizam-se as variáveis de saída do programa de acordo com o estado actual da máquina de estados. Exemplo[1]:

Listagem 6: Programa PLC relativo à Figura 5.3.

```
1.    PROGRAM MAIN
2.    VAR
3.          Estado           : INT := 0; (* Variável de estado  *)
4.          LigarInterruptor : BOOL;     (* Variável de entrada *)
5.          LuzAcesa         : BOOL;     (* Variável de saída   *)
6.    END_VAR
7.
8.    (* Evolução normal da máquina de estados *)
9.    IF (Estado = 0) AND (LigarInterruptor) THEN
10.         Estado := 1;
11.   ELSIF (Estado = 1) AND ( LigarInterruptor = FALSE) THEN
12.         Estado := 0;
13.   END_IF
14.
15.   (* Atribuição de saídas da máquina de estados *)
16.   LuzAcesa := (Estado = 1);
17.
18.   END_PROGRAM
```

Note-se que a variável `Estado` é inicializada a zero, que corresponde ao estado em que a luz se encontra apagada. Quando o sistema se encontra no estado zero, e  o interruptor é ligado,

1 O exemplo apresentado é meramente académico, pois poderia ser facilmente representado da seguinte maneira: “LuzAcesa := LigarInterruptor;”.

o estado do sistema muda para 1. E quando o estado é 1 e o interruptor é desligado, o sistema volta ao estado zero. A variável `LuzAcesa` toma o valor verdadeiro quando o sistema se encontra no estado 1. E assim se mostra um exemplo de como desenvolver um programa PLC que controle um sistema. Claro que este método só funciona com sistemas cujo comportamento possa ser representado por uma máquina de estados. No caso de um grafcet, cada situação possível seria um estado diferente, e para grafcets complexos há uma explosão de estados que torna impossível o desenvolvimento de programas PLC por este método. Sem contar que o GRAFCET introduz transições temporizadas, o que impossibilitaria ainda mais a utilização deste método. Da mesma forma, as redes de Petri podem ter vários lugares activos simultâneamente e com um número variado de marcas. Assim, é necessário criar estruturas de dados que representem fielmente um grafcet ou uma rede de Petri. O próximo capítulo irá, inicialmente, apresentar a arquitectura a aplicação WEBGRAF 2, e depois irá apresentar a arquitectura do código gerado pela aplicação que executa modelos descritos em grafcets ou redes de Petri.

# 6 Arquitectura da aplicação WEBGRAF 2 e do código gerado

O presente capítulo destina-se a apresentar: a arquitectura geral da aplicação WEBGRAF 2; as tecnologias usadas para o seu desenvolvimento; a metodologia de conversão do modelo de controlo para um programa PLC – metodologia síncrona; e a estrutura do código gerado pelo tradutor, tanto para as redes de Petri como para o GRAFCET. Irá primeiramente ser apresentada uma esquematização geral das diferentes fases da exploração da aplicação, desde a explicitação do modelo de controlo sequencial até à obtenção do código de programação, e, posteriormente, cada uma das fases será examinada individualmente. Também irão ser apresentadas as diferentes metodologias de conversão de modelos de controlo para programas PLC, e justificada a razão pela escolha da metodologia síncrona. Finalmente, é explicada a estrutura do código gerado.

## 6.1 Arquitectura geral

A aplicação WEBGRAF 2 pode ser executada em dois modos distintos: modo *standalone* e modo Cliente-Servidor. As sub-secções seguintes apresentam a arquitectura geral da aplicação em cada um destes modos.

### 6.1.1 Modo Servidor-Cliente

A arquitectura Cliente-Servidor é uma arquitectura onde participam dois tipos de entidades: os servidores (fornecedores de serviços) e os clientes (que solicitam os serviços dos servidores). Geralmente, estas duas entidades residem em hardwares separados. Um cliente faz um pedido a um servidor e espera pela resposta, enquanto que o servidor espera por pedidos, processa-os e retorna o resultado ao cliente. O pedido é feito por um programa cliente, que é um

programa em execução no computador do utilizador para comunicar com o servidor. No caso da aplicação WEBGRAF 2, o programa cliente é qualquer navegador web (Firefox, Internet Explorer, etc.).

A Figura 6.1 apresenta a arquitectura da aplicação WEBGRAF 2 no modo Cliente-Servidor.

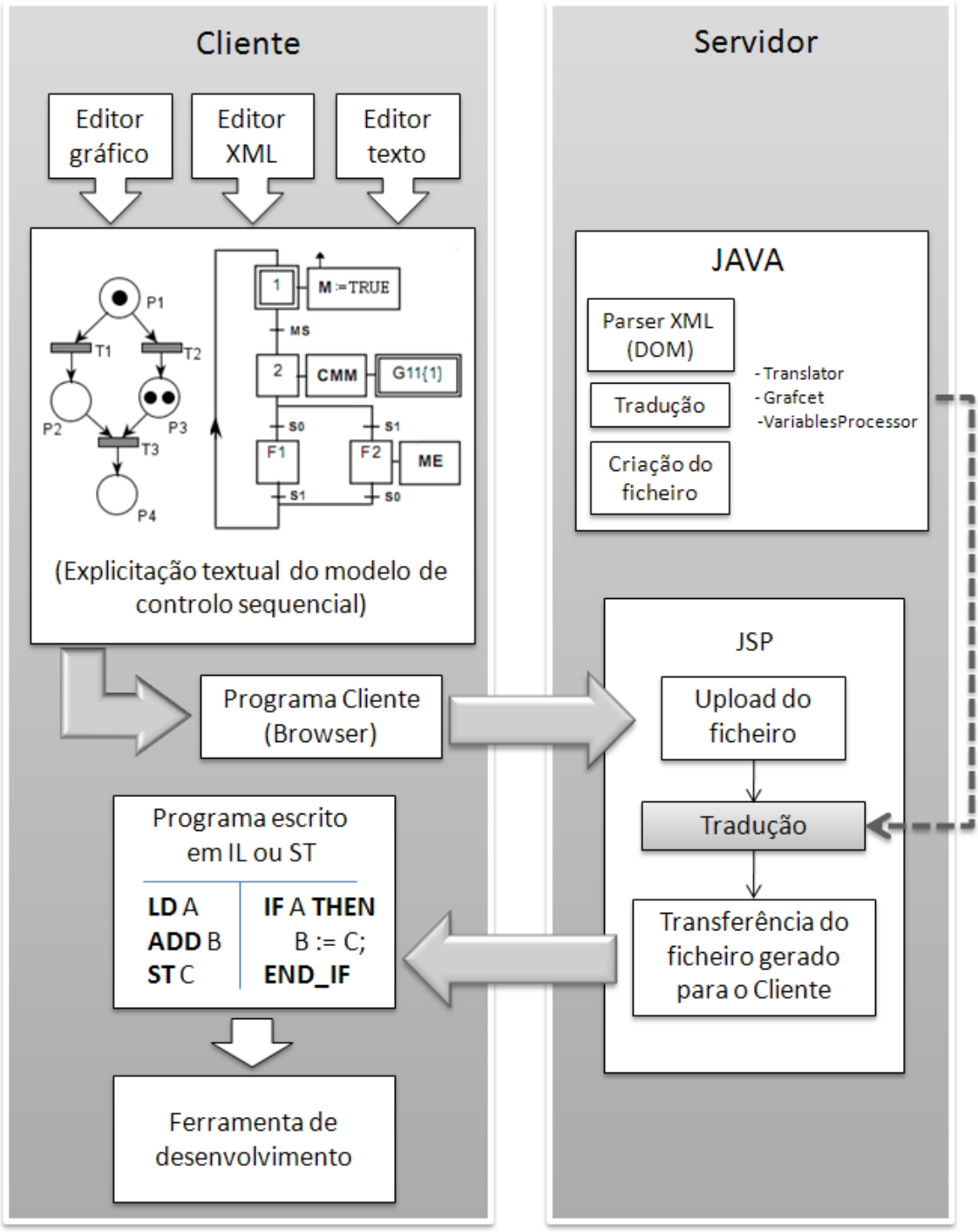



Figura 6.1: Arquitectura geral da aplicação WEBGRAF 2.

### 6.1.2 Modo Standalone

Uma aplicação *standalone* é uma aplicação que não tem dependências externas, e que pode executar na máquina do utilizador em modo offline (ao contrário duma aplicação Cliente-Servidor).

Na Figura 6.2 está ilustrada a arquitectura da aplicação *standalone*.

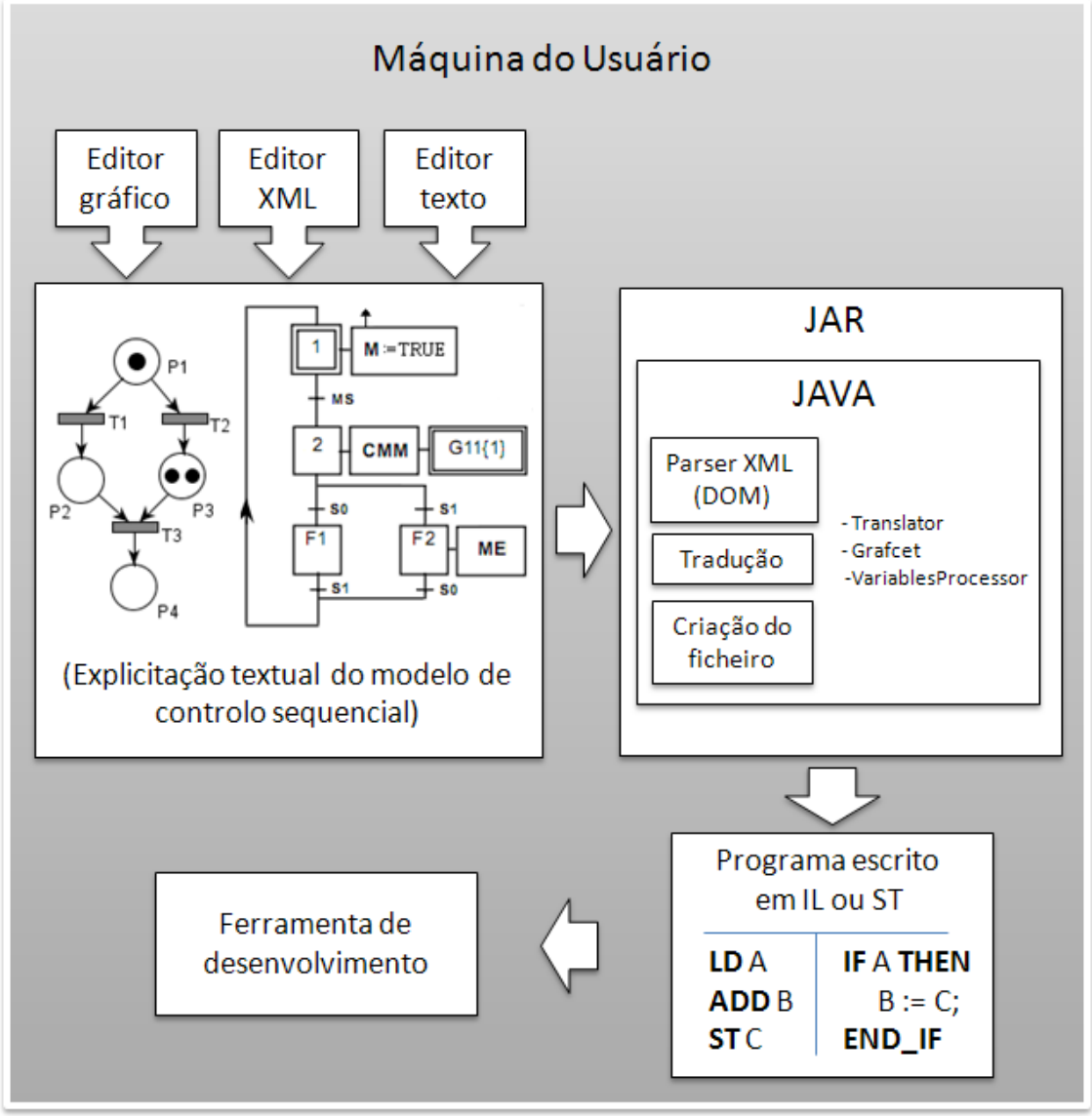



Figura 6.2: Arquitectura geral da aplicação WEBGRAF 2 - modo standalone.

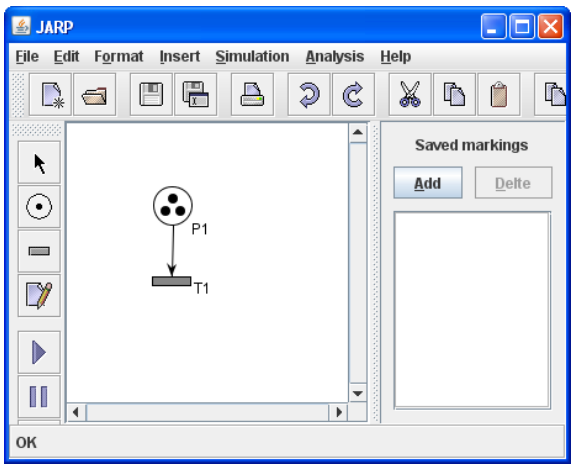
Antes de passar à apresentação detalhada de cada uma das fases do processo de tradução, convém mencionar que o tradutor foi desenvolvido em JAVA. O modo *standalone* consiste num ficheiro JAR que encapsula as classes do tradutor, enquanto que o programa servidor utiliza essas classes recorrendo a javaBeans.

## 6.2 Fases da arquitectura

### 6.2.1 Explicitação dos modelos de controlo sequencial

A explicitação dos modelos de controlo sequencial é feita com as linguagens PNML, no caso das redes de Petri, e GML, no caso dos GRAFCETs. Os ficheiros que contêm a explicitação dos modelos podem ser criados com editores básicos de texto, como por exemplo o Notepad do Windows ou o Notepad++. Alternativamente, podem também ser criados recorrendo a editores de XML, como por exemplo o Microsoft XML NotePad 2007. A aplicação WEBGRAF 2 disponibiliza uma versão da ferramenta JARP que poderá ser usada para composição de redes de Petri e geração da respectiva explicitação textual. Esta solução torna a criação da explicitação textual numa tarefa menos ingrata e menos sujeita a erros.

A Figura 6.3 mostra a criação de uma rede de Petri, com a ferramenta JARP, e um excerto do ficheiro PNML gerado.




```
<place id="p1">
      <name>
            <text>P1</text>
      </name>
      <initialMarking>
            <text>3</text>
      </initialMarking>
</place>
<transition id="t1">
      <name>
            <text>T1</text>
      </name>
</transition>
```

Figura 6.3: Geração da explicitação textual de uma rede de Petri com a ferramenta JARP.

A descrição das várias funcionalidades da ferramenta JARP poderá ser encontrada na sua homepage [JARP01].

### 6.2.2 JavaServer Pages (JSP)

O programa servidor foi desenvolvido com a tecnologia JavaServer Pages (JSP). Este é responsável por:

- fazer o *upload* do ficheiro com a especificação do modelo de controlo sequencial;
- invocar os métodos do tradutor para traduzir e gerar o ficheiro do programa;

- retornar o ficheiro gerado ao cliente.

O programa servidor vê o tradutor como uma caixa-branca, e que utiliza para gerar o ficheiro com o programa do respectivo modelo de controlo sequencial.

A Figura 6.4 apresenta a arquitectura do programa servidor.

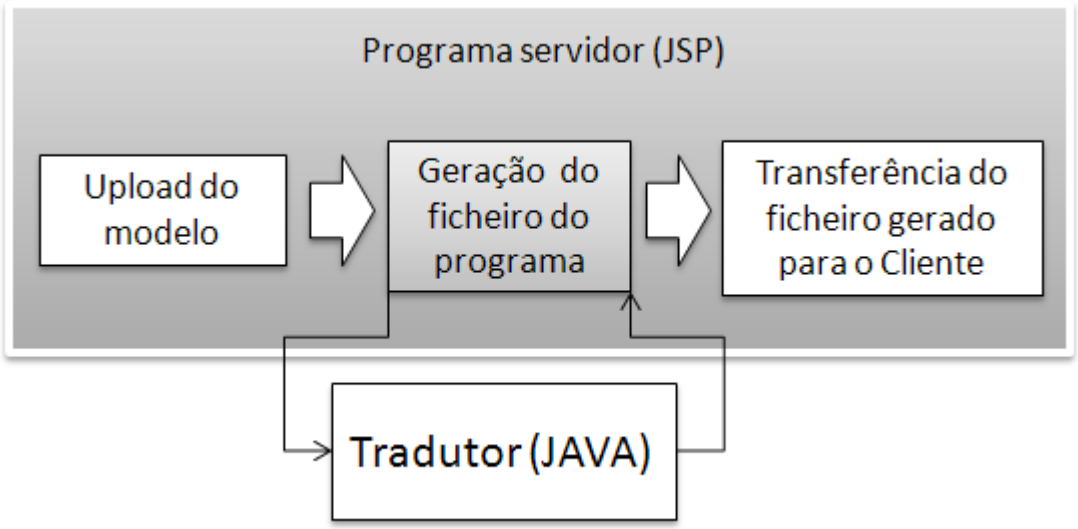



Figura 6.4: Arquitectura do programa servidor.

### 6.2.3 Tradutor JAVA

O tradutor é constituído por 3 classes JAVA:

- Translator.class
- Grafcet.class
- VariablesProcessor.class

A classe Translator é responsável por fazer o *parsing* do ficheiro da especificação do modelo e criação em memória duma representação da estrutura do ficheiro, que será analisada na secção seguinte. Se o ficheiro corresponder a um documento PNML, a classe Translator realiza a tradução e respectiva criação do ficheiro do programa, se corresponder a um documento GML, a tradução é feita pela classe Grafcet. A classe VariablesProcessor executa funções de auxílio à análise da representação em memória do documento.

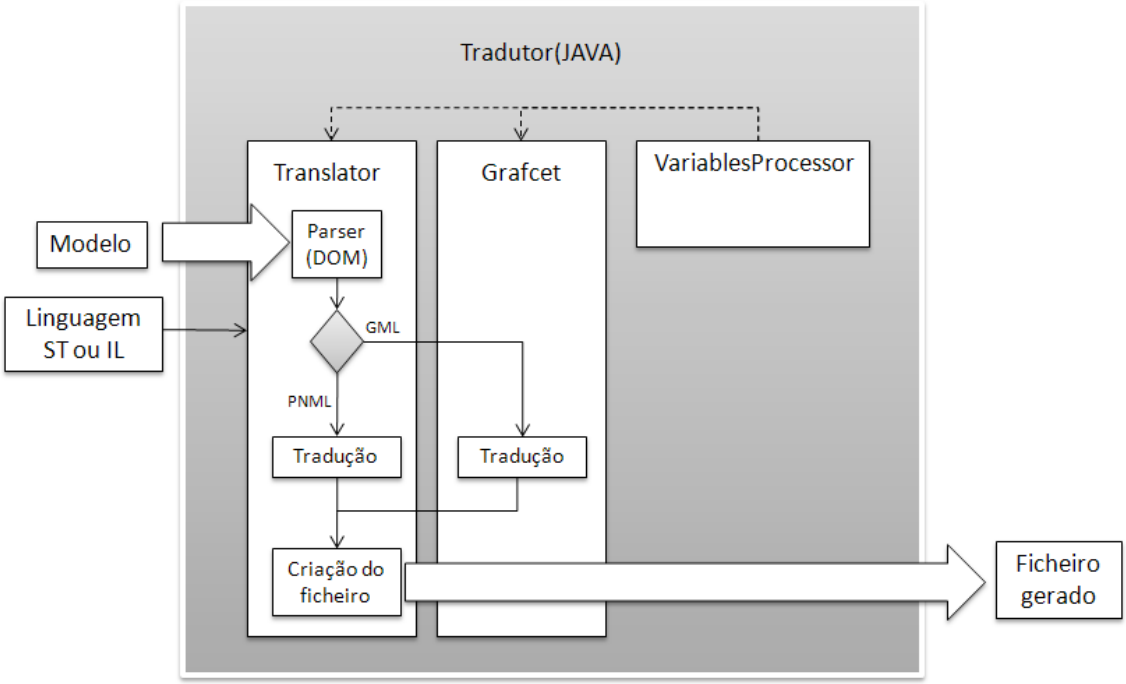



Figura 6.5: Arquitectura do tradutor.

### 6.2.4 Parser XML

Um *parser* XML é um processador que analisa um documento XML e obtêm a informação desse documento de uma forma estruturada, para depois passá-la para uma aplicação.

Dentro dos vários *parsers* XML disponíveis, o escolhido foi o JAXP (Java for XML Processing). JAXP usa uma interface do tipo DOM (Document Object Model) [DOM00], que constrói uma representação, do documento XML, estruturada em árvore.

## 6.3 Método síncrono

Tipicamente, a conversão de um modelo de controlo sequencial num programa PLC é feita segundo um de dois métodos: síncrono e assíncrono. No método síncrono, considera-se que o sistema é regido por um sinal de relógio, e, a cada transição desse sinal, o sistema evolui de um estado anterior para um estado actual. Portanto, o estado actual é obtido em função do estado anterior e das entradas actuais. As saídas do sistema vão sendo obtidas em função do estado actual. No método assíncrono, considera-se que o sistema reage à alteração lógica de uma variável. Consequentemente, a alteração lógica de uma variável, pode levar à actualização de outras variáveis que estejam dependentes dela, o que pode resultar num encadeamento sequencial de actualização de variáveis, até que seja atingida uma situação estável.

Vale a pena relembrar que um PLC corre um programa de modo cíclico, actualizando as variáveis de entrada imediatamente antes do ciclo começar, e fazendo a actualização das variáveis de saída no final de cada ciclo. Assim, pode-se considerar que o método síncrono é o

mais ajustado aos varrimentos cíclicos de um PLC. Isto porque, este método encara cada ciclo do PLC como um sinal de relógio, e, em cada ciclo, o sistema passa de um estado anterior para o estado actual, ou seja, determina quais as transições disparáveis, guardando o estado da rede resultante numa rede auxiliar (que corresponde ao estado actual), e no final do ciclo a rede original (que corresponde ao estado anterior) é actualizada com os valores da rede auxiliar. O método assíncrono, por seu lado, dispara todas as transições possíveis em cada ciclo.

Apesar de um programa assíncrono ser geralmente mais rápido do que um síncrono, ele não lida tão bem com cenários instáveis.

Para uma melhor compreensão destes dois métodos analise-se agora a seguinte figura:

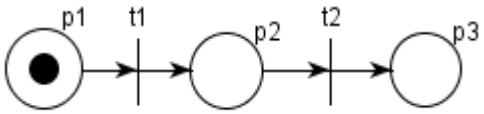



Figura 6.6: Rede de Petri para comparação dos métodos síncrono e assíncrono.

Assumindo que $t_1$ e $t_2$ são verdadeiros, o método síncrono, por definição, faz o sistema passar de um estado anterior, $p_1$, para um estado actual, $p_2$, num ciclo, e, no ciclo seguinte,.faz o sistema passar dum estado anterior, que agora é $p_2$, para um estado actual, $p_3$. Ou seja, este método faz a marca saltar de $p_1$ para $p_2$ num ciclo e de $p_2$ para $p_3$ noutro ciclo. O método assíncrono faz a marca saltar directamente de $p_1$ para $p_3$ no mesmo ciclo, não chegando a actualizar devidamente $p_2$.

Dois dois métodos apresentados nesta secção, o usado pela aplicação WEBGRAF 2 é o síncrono.

## 6.4 Arquitectura do código gerado para redes de Petri

Os elementos das Tabelas 3.1 e 3.2 (da secção 3.2) que possuem uma cruz (“X”), na última coluna da tabela, são os que estão contemplados no WEBGRAF 2.

O código gerado pelo tradutor é constituído por: duas *structs*, uma declaração de variáveis globais, oito *Function Blocks* e um *Program.* Basicamente, as *struct* são estruturas de dados usadas para representar transições e lugares; nas variáveis globais estão incluídos, entre outros, dois *arrays:* um para guardar todos os lugares da rede, e o outro para guardar as transições; o *program* executa o algoritmo do método síncrono; e as diferentes *Function Blocks* são usadas pelo *program* para modificar o estado da rede, verificar a receptividade de uma transição, etc. A ordem em que as transições e lugares são guardados nos seus 0próprios *arrays* é a mesma ordem em que surgem no documento PNML (ver Figura 6.7). O índice de ambos os *array* começa em 1. O tamanho do *array* das transições é, logicamente, igual ao número de transições existentes na rede , e o tamanho do *array* de lugares é igual ao número de lugares existentes.

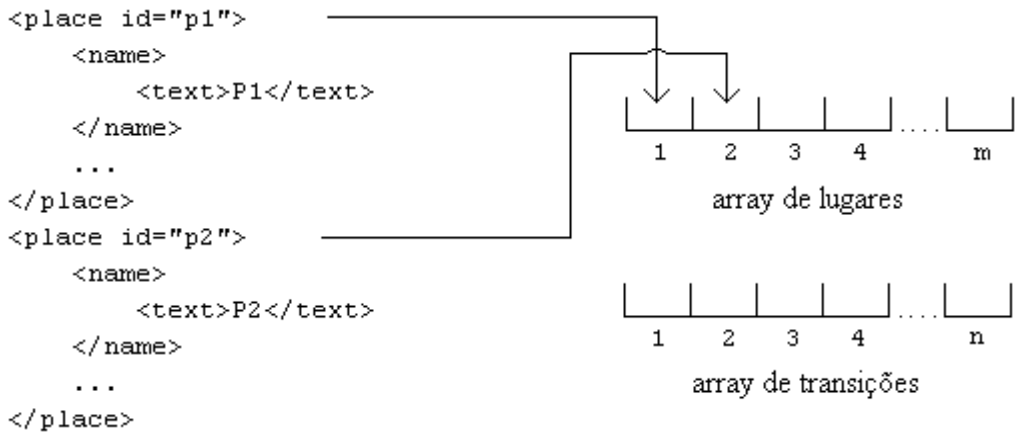



Figura 6.7: Ordem da posição dos lugares no *array*.

O nome de cada *FB* e as respectivas variáveis de entrada e saída estão apresentados na tabela seguinte:

Tabela 6.1: Lista das *Function Blocks* geradas pelo tradutor.

| **Function Block** | **Variáveis de entrada** | **Variáveis de saída** |
|---|---|---|
| IS_READY_TO_FIRE | transition : UINT | ready : BOOL |
| FIRE_TRANSITION | transition : UINT | |
| CONDITION_OF_TRANSITION | transition : UINT | result : BOOL |
| ADD_TOKENS | place : UINT | tokens : UINT |
| REMOVE_TOKENS | place : UINT<br>tokens : UINT | |
| TIMER_OF_TRANSITION | transition : UINT<br>IN : BOOL | Q : BOOL |
| UPDATE_OUTPUTS | place : UINT | |
| RE_TRIG | CLK : BOOL | Q : BOOL |

A variável *transition* é comum a várias *FBs.* Esta variável é do tipo *unsigned int* (inteiro não negativo) e corresponde a um índice do *array* de transições. Ou seja, representa a transição que está nessa posição do *array*. Passando a explicar com um exemplo: a seguinte invocação FIRE_TRANSITION(1), resultava no disparo da transição situada na primeira posição do *array*. A variável de entrada *place* corresponde a um índice do *array* de lugares, e representa o lugar que está nessa posição do *array*.

De forma a demonstrar o funcionamento de cada unidade de código, gerada pelo tradutor, irão ser apresentados excertos do código gerado. Como tal, considere-se uma célula de fabrico flexível constituída por: um *buffer* de peças à entrada, um braço robótico e uma máquina de produção e *buffer* de peças à saída. As peças são colocadas no *buffer* de entrada por acção humana, o braço robótico transporta uma peça peça de cada vez do *buffer* de entrada para a máquina de produção. O braço robótico demora 10 segundos a realizar o transporte de uma peça

para a máquina e 5 segundos a voltar ao *buffer* de entrada para ficar pronto a transportar a próxima peça. A máquina de produção leva 30 segundos a processar a peça e transformá-la em algo novo, as As peças finais são colocadas no *buffer* de saída e podem ser removidas por acção humana. O funcionamento desta célula de fabrico flexível está modelizada através da rede de Petri da Figura 6.8.

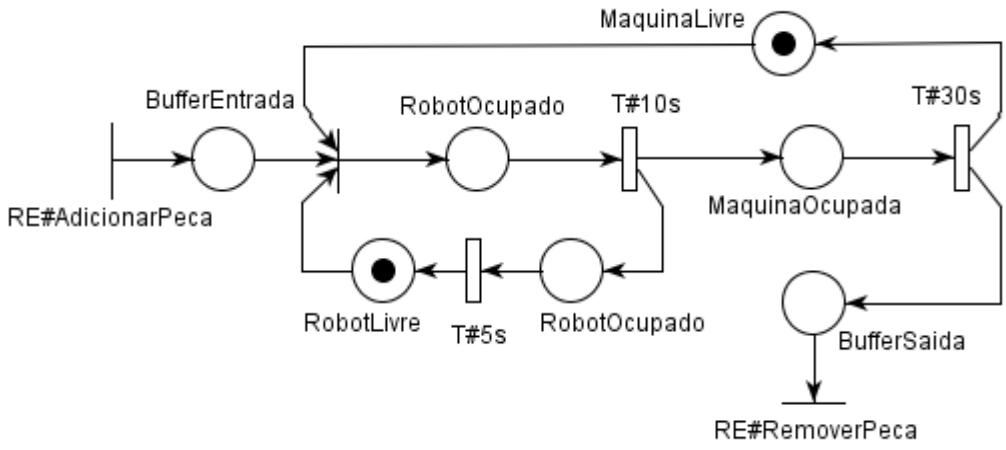



Figura 6.8: Modelação da célulo de fabrico flexível em redes de Petri.

As variáveis de saída do sistemas são:

- **BufferEntrada –** indica se há peças disponíveis no *buffer* de entrada;
- **RobotOcupado –** indica se o braço robótico está em funcionamento;
- **RobotLivre –** indica se o braço robótico está disponível para carregar nova peça;
- **MaquinaLivre –** indica se a máquina está disponível para receber nova peça;
- **MaquinaOcupada –** indica se a máquina está actualmente a trabalhar;
- **BufferSaida –** indica se há peçasno *buffer* de saída prontas a serem retiradas.

As variáveis de entrada são:

- **AdicionarPeca** – uma mudança de sinal de 0 para 1 desta variável leva à adição de uma marca no lugar de saída da respectiva transição;
- **RemoverPeca** – uma mudança de sinal de 0 para 1 desta variável levar à remoção de uma marca no lugar de entrada da respectiva transição.

As sub-secções seguintes destinam-se a apresentar cada uma das unidades de código geradas pelo tradutor, explicando o seu funcionamento e apresentando excertos do respectivo código quando justificado. O código apresentado corresponde à tradução em linguagem ST. Todos os exemplos são relativos à rede da Figura 6.8.

### 6.4.1 VAR_GLOBAL

Na declaração das variáveis globais estão incluídas:

- Estrutura de dados para representar os lugares;
- Estrutura de dados para representar as transições;
- Variáveis de entrada;
- Variáveis de saída;
- Estrutura de dados que representa os lugares da rede auxiliar;
- Estrutura de dados que representa as transições da rede auxiliar.

De seguida é apresentado um excerto do código gerado pelo tradutor, que corresponde à declaração das variáveis globais.

Listagem 7: Código gerado respeitante à declaração das variáveis globais.

```
1.  VAR_GLOBAL
2.        PLACES      : ARRAY [1..7] OF PLACE;
3.        TRANSITIONS : ARRAY [1..6] OF TRANSITION;
4.
5.        (* NUMBER OF TRANSITIONS *)
6.        TRANS_LENGTH : UINT := 6;
7.
8.        (* OUTPUTS *)
9.        BufferEntrada : BOOL;
10.       RobotOcupado : BOOL;
11.       RobotOcupado : BOOL;
12.       MaquinaOcupada : BOOL;
13.       MaquinaLivre : BOOL := TRUE;
14.       RobotLivre : BOOL := TRUE;
15.       BufferSaida : BOOL;
16.
17.       (* INPUTS *)
18.       ADICIONARPECA : BOOL;
19.       REMOVERPECA : BOOL;
20.
21.       (* virtual net *)
22.       PLACES_TEMP : ARRAY [1..7] OF PLACE;
23.       TRANSITIONS_TEMP : ARRAY [1..6] OF TRANSITION;
24. END_VAR
```

O código apresentado não possui a inicialização dos *arrays* para uma melhor perceptibilidade do código.

PLACES e TRANSITIONS são *arrays* que guardam variáveis do tipo PLACE e TRANSITION, respectivamente. O tipo de dados tanto de PLACE como de TRANSITION é do tipo *struct*. As declarações destes tipos serão apresentadas nas secções 6.4.2 e 6.4.3. Estes *arrays* guardam o estado da rede e são declarados globalmente para que a informação da rede possa ser lida e modificada em qualquer parte do código. O *array* PLACE guarda todos os lugares da rede, e o *array* TRANSITIONS guarda todas as transições. A ordem em que são guardados no *array* corresponde à ordem em que aparecem no documento PNML. A variável TRANS_LENGTH, do tipo UINT, guarda o número total de transições da rede.

A rede de Petri apresentada Figura 6.8 contém dois lugares que, inicialmente, contêm uma marca cada um. Das variáveis de saída apresentadas na Listagem 7, note-se que MaquinaLivre e RobotLivre são inicializadas a *true*, que são as variáveis associadas aos lugares com marcação inicial. As restantes são inicializadas a *false*, pois declarar uma variável booleana sem indicar o seu valor de inicialização, corresponde a inicializá-la com *false*.

A seguir aparecem as variáveis de entrada, ADICIONARPECA e REMOVERPECA, que correspondem às variáveis usadas na condições das transições.

E, finalmente, a declaração das variáveis globais é também constituída pelas variáveis PLACES_TEMP e TRANSITIONS_TEMP, que são *arrays* que guardam variáveis do tipo PLACE e TRANSITION que representam os lugares e transições da rede auxiliar.

### 6.4.2 Struct – PLACE

O tipo de dados criado para representar um lugar é *struct*. A *struct* contém uma variável inteira não negativa, "Marking", que guarda o número de marcas que o lugar possui. O código gerado correspondente à declaração da *struct* PLACE é o seguinte:

Listagem 8: Declaração da *Struct* PLACE.

```
1.  TYPE
2.   PLACE : STRUCT
3.       Marking : UINT;
4.   END_STRUCT
5.  END_TYPE
```

As variáveis que representam os lugares da rede da Figura 6.8 seriam inicializadas da seguinte forma:

Listagem 9: Inicialização de uma variável do tipo PLACE.

```
1.  variavelLugar : PLACE :=
2.  (
3.      Marking:=0
4.  );
```

`Marking` inicializado a 0 para os lugares que não contêm marcas, e inicializado a 1 para os que contêm uma marca.

### 6.4.3 Struct – TRANSITION

O tipo de dados criado para representar uma transição também é uma *struct.* Esta contém quatro *arrays* de `UINT`, e duas variáveis do tipo `UINT`.

Listagem 10: Declaração da *Struct* TRANSITION.

```
1.  TYPE
2.        TRANSITION : STRUCT
3.              Source        : ARRAY [1..3] OF UINT;
4.              Target        : ARRAY [1..2] OF UINT;
5.              Source_weight : ARRAY [1..3] OF UINT;
6.              Target_weight : ARRAY [1..2] OF UINT;
7.              Source_len    : UINT;
8.              Target_len    : UINT;
9.        END_STRUCT
10. END_TYPE
```

O *array* Source contém os índices do *array* `TRANSITIONS`, declarado nas variáveis globais, que correspondam aos seus lugares de entrada. Ou seja, é um *array* com referências para os lugares da rede que sejam seus lugares de entrada. O seu tamanho é o número máximo de lugares de entrada que uma transição tem na rede, que no caso da rede apresentada na Figura 6.8 é 3. A variável `Source_len` guarda o número de lugares de entrada que a transição possui. O *array* `Target` guarda os índices do *array* `TRANSITIONS` que correspondam aos seus lugares de saída. O seu tamanho corresponde, uma vez mais, ao número máximo de lugares de saída que uma transição possa ter – 2 no caso da rede apresentada na Figura 6.8. A variável `Targe_len` contém o número de lugares de saída que a transição possui.

Source_weight e `Target_weight` são *arrays que* contêm os pesos dos arcos que ligam a transição a cada um dos lugares referenciados por `Source` e `Target`, respectivamente. Cada posição n dos *arrays* dos pesos corresponde à posição n dos *arrays* dos lugares. Ou seja, `Source_weight[1]` é o peso do arco que liga a transição ao lugar referenciado por `Source[1]`.

A variável que representa a transição, da rede da Figura 6.8, com etiqueta `T#10s` seria inicializada da seguinte forma:

Listagem 11: Inicialização de uma variável do tipo TRANSITION.

```
1. variavelTransicao : TRANSITION :=
2. (
3.       Source          := [2,0,0],
4.       Target          := [3,4],
5.       Source_weight   := [1,0,0],
```

```
6.       Target_weight   := [1,1],
7.       Source_len      := 1,
8.       Target_len      := 2
9. );
```

### 6.4.4 Program – MAIN_PROGRAM

O *program* corresponde à parte principal do programa gerado. É esta a parte do código que é executada ciclicamente pelo PLC, ou pelo programa de desenvolvimento. No início de cada ciclo do *program* são lidas as variáveis de entrada e no final de cada ciclo as variáveis de saída são actualizadas. O funcionamento geral do *program* caracteriza-se por verificar quais as transições que podem disparar e criar uma representação temporária do novo estado da rede com as transições disparadas. E, depois disso, actualizar a rede original tornando-a igual à temporária. A verificação das condições das transições e o disparo das mesmas são realizados através da chamada a *Function Blocks*.

Listagem 12: Código correspondente ao *program*.

```
1. PROGRAM MAIN_PROGRAM
2. VAR
3.  IS_READY_TO_FIRE    : IS_READY_TO_FIRE;
4.  FIRE_TRANSITION     : FIRE_TRANSITION;
5.  transition          : UINT := 1;
6.  ready               : BOOL;
7. END_VAR
8.
9. (* fire enabled transitions *)
10. FOR transition := 1 TO TRANS_LENGTH DO
11.       IS_READY_TO_FIRE(transition:=transition,
    ready=>ready);
12.       IF ready THEN
13.             FIRE_TRANSITION(transition := transition);
14.       END_IF
15. END_FOR
16.
17. (* update net *)
18. TRANSITIONS := TRANSITIONS_TEMP;
19. PLACES := PLACES_TEMP;
20. END_PROGRAM
```

IS_READY_TO_FIRE é uma *Function Block* que verifica, no momento em que é chamada, se uma dada transição está pronta a disparar, recebendo como parâmetro de entrada `transition`, e retornando como valor de saída `ready`. Ou seja, verifica se a sua condição é

verdadeira ou, no caso de ser uma transição temporizada, se o seu temporizador já atingiu o tempo necessário para poder disparar. Esta verificação é feita para todas as transições através do ciclo FOR: a variável `transition`, declarada localmente, toma os valores entre 1 e `TRANS_LENGTH`, que correspondem todos os índices de `TRANSITIONS`, e `IS_READY_TO_FIRE` é chamado para cada um desses valores. A variável `ready` é então actualizada com o valor de saída (cujo nome também é `ready`) de `IS_READY_TO_FIRE`. Se uma transição estiver pronta a ser disparada é feita uma chamada à *Function Block* `FIRE_TRANSITION`. Este disparo ocorre na rede auxiliar e não na original. Finalmente, a rede original é actualizada com os valores da rede auxiliar, como se pode verificar nas linhas 18 e 19 da Listagem 12.

### 6.4.5 Function Block – IS_READY_TO_FIRE

Listagem 13: Código correspondente à *Function Block* IS_READY_TO_FIRE.

```
1.  FUNCTION_BLOCK IS_READY_TO_FIRE
2.  VAR_INPUT
3.     transition  : UINT;
4.  END_VAR
5.  VAR_OUTPUT
6.      ready : BOOL;
7.  END_VAR
8.  VAR
9.      TIMER_OF_TRANSITION : TIMER_OF_TRANSITION;
10.     CONDITION_OF_TRANSITION : CONDITION_OF_TRANSITION;
11.     I: UINT;
12.     K: UINT;
13.     result : BOOL;
14.     IN : BOOL;
15.  END_VAR
16.
17.  IN := FALSE;
18.  CONDITION_OF_TRANSITION(transition := transition, result =>
result);
19.  IF result THEN
20.      (* checks whether all input places have enough tokens *)
21.      K := TRANSITIONS[transition].Source_len;
22.      FOR I := 1 TO K DO
23.         IF  PLACES[TRANSITIONS[transition].Source[I]].Marking <
TRANSITIONS[transition].Source_weight[I] THEN
24.            TIMER_OF_TRANSITION(transition := transition, IN :=
IN, Q => ready);
25.              RETURN;
26.          END_IF
27.      END_FOR
```

```
28.      IN := TRUE;
29.  END_IF
30.
31.   TIMER_OF_TRANSITION(transition := transition, IN := IN, Q =>
ready);
32.
33.  (* resets the timer if it already reached the desired value*)
34.  IF ready = TRUE THEN
35.          TIMER_OF_TRANSITION(transition := transition, IN :=
FALSE);
36.  END_IF
37.   END_FUNCTION_BLOCK
```

Esta *Function Block* recebe como parâmetro de entrada um índice do *array* global TRANSITIONS, `transition`, e verifica se essa transição está pronta ser disparada. A variável de saída *ready* toma o valor de *true* ou *false* conforme a transição pode ou não disparar no momento em que a *Function Block* é chamada. Uma transição pode disparar se os seus lugares de entrada têm pelo menos o mesmo número de marcas que o peso do arco que o liga à transição e se a condição da transição for verdadeira ou, no caso de ser uma transição temporizada, o temporizador ter atingido o tempo desejado.

### 6.4.6 Function Block – FIRE_TRANSITION

Irá seguidamente ser apresentado o código da Function Block FIRE_TRANSITION.

Listagem 14: Function Bock FIRE_TRANSITION

```
1.   FUNCTION_BLOCK FIRE_TRANSITION
2.   VAR_INPUT
3.      transition      : UINT;
4.   END_VAR
5.   VAR
6.      UPDATE_OUTPUTS  : UPDATE_OUTPUTS;
7.      REMOVE_TOKENS   : REMOVE_TOKENS;
8.      ADD_TOKENS      : ADD_TOKENS;
9.      counter         : UINT := 1;
10.     place           : UINT;
11.     tokens          : UINT;
12.     source_len      : UINT;
13.     target_len      : UINT;
14.  END_VAR
15.
16.  (* removes tokens from input places *)
17.  source_len := TRANSITIONS[transition].Source_len;
18.  FOR counter := 1 TO source_len DO
19.     place := TRANSITIONS[transition].Source[counter];
20.     tokens := TRANSITIONS[transition].Source_weight[counter];
```

```
21.    REMOVE_TOKENS(place := place, tokens := tokens);
22.    UPDATE_OUTPUTS(place := place);
23.  END_FOR
24.
25.  (* adds tokens to the output places *)
26.  target_len := TRANSITIONS[transition].Target_len;
27.  FOR counter := 1 TO target_len DO
28.    place := TRANSITIONS[transition].Target[counter];
29.    tokens := TRANSITIONS[transition].Target_weight[counter];
30.    ADD_TOKENS(place := place, tokens := tokens);
31.    UPDATE_OUTPUTS(place := place);
32.  END_FOR
33.  END_FUNCTION_BLOCK
```

Basicamente, o que esta Function Block faz é consumir as marcas dos lugares de entrada de uma transição e colocar as devidas marcas nos lugares de saída.

O primeiro ciclo FOR percorre o *array* `Source da` transição passada como argumento, e obtém os índices do array global TRANSITIONS correspondentes aos seus lugares de entrada – `place`, e, para cada lugar de entrada, obtém o número de marcas a remover – `tokens`. Finalmente, procede à remoção das marcas, `REMOVE_TOKENS`, e actualização das variáveis de saída, `UPDATE_OUTPUTS`, esta actualização é necessária para colocar a variável de saída associada ao lugar de entrada a *false*, caso ele fique sem marcas. O segundo ciclo FOR percorre o *array* `Target da transição passada` como argumento, e obtém os índices do *array* global TRANSITIONS correspondentes aos seus lugares de saída – `place`. Obtém também o número exacto de marcas a adicionar ao respectivo lugar de saída – `tokens`. Finalmente, procede à adição das marcas nos respectivos lugares, `ADD_TOKENS`, e actualização das variáveis de saída, para colocar a variável de saída associada ao lugar de saída a *true*, caso esta não tivesse marcas e passasse a ter. As *Function Blocks* `REMOVE_TOKENS`, `ADD_TOKENS` e `UPDATE_OUTPUTS` serão analisadas em secções posteriores.

### 6.4.7 Function Block – CONDITION_OF_TRANSITION

Listagem 15: Function Bock CONDITION_OF_TRANSITION

```
1.  FUNCTION_BLOCK CONDITION_OF_TRANSITION
2.  VAR_INPUT
3.     transition : UINT;
4.  END_VAR
5.  VAR_OUTPUT
6.     result : BOOL;
7.  END_VAR
8.  VAR
9.      RE_ADICIONARPECA_1 : RE_TRIG;
10.     RE_REMOVERPECA_6 : RE_TRIG;
11.  END_VAR
```

```
12.
13.  CASE transition OF
14.      1:
15.            RE_ADICIONARPECA_1(CLK := ADICIONARPECA);
16.            result := RE_ADICIONARPECA_1.Q ;
17.      2:
18.            result := TRUE;
19.      3:
20.            result := TRUE;
21.      4:
22.            result := TRUE;
23.      5:
24.            result := TRUE;
25.      6:
26.            RE_REMOVERPECA_6(CLK := REMOVERPECA);
27.            result := RE_REMOVERPECA_6.Q ;
28.      ELSE
29.          result := FALSE;
30.  END_CASE
31.   END_FUNCTION_BLOCK
```

A *Function Block* CONDITION_OF_TRANSITION tem como variável de entrada `transition` que corresponde ao índice da transição do *array* global TRANSITIONS. Esta *FB* (*Function Block*) verifica se a condição da transição é verdadeira ou falsa no momento em que é chamada, e a variável de saída `result` toma um dos valores *true* e *false* de acordo com o resultado da condição.

Note-se que esta FB retorna o valor de *true* para transições temporizadas.

### 6.4.8 Function Block – ADD_TOKENS

Esta *FB* adiciona um determinado número de marcas a um lugar. Ela recebe como variáveis de entrada: `place` e `tokens`, que correspondem ao índice do lugar no *array* global PLACES_TEMP e ao número de marcas a ser adicionado, respectivamente. A adição de marcas é feita na rede auxíliar, daí o *array* PLACES_TEMP ao invés de PLACES.

Listagem 16: Function Bock ADD_TOKENS

```
1.  FUNCTION_BLOCK ADD_TOKENS
2.  VAR_INPUT
3.     place  : UINT;
4.     tokens : UINT;
5.  END_VAR
6.
7.    PLACES_TEMP[place].Marking  :=  PLACES_TEMP[place].Marking  +
tokens;
8.  END_FUNCTION_BLOCK
```

### 6.4.9 Function Block – REMOVE_TOKENS

A função da *FB* REMOVE_TOKEN permite remover um determinado número de marcas de um lugar da rede auxiliar. O lugar é dado pela variável de entrada `place`, que corresponde ao índice no *array* `PLACES_TEMP`, e o número de marcas a remover é dado pela variável de entrada `tokens`.

Listagem 17: Function Bock REMOVE_TOKENS

```
1. FUNCTION_BLOCK REMOVE_TOKENS
2. VAR_INPUT
3.    place : UINT;
4.    tokens : UINT;
5. END_VAR
6.
7. PLACES_TEMP[place].Marking := PLACES_TEMP[place].Marking-tokens;
8. PLACES[place].Marking := PLACES[place].Marking - tokens;
9. END_FUNCTION_BLOCK
```

Também é removida a mesma quantidade de marcas ao lugar da rede original, para evitar situações inesperadas em caso de redes erróneas. Imagine-se um lugar com uma marca, sendo o lugar de entrada de duas transições diferentes, cada uma ligada por um arco de peso unitário. Se as condições de ambas as transições estivessem a *true*, ambas as transições iriam disparar caso as marcas não fossem removidas da rede original, pois após umas delas disparar, as marcas iriam continuar visíveis para a outra. Claro que situações destas só acontecem no caso de um desenho da rede errado por parte do utilizador, mas em todo o caso convém evitar estas situações que levem a estados incoerentes.

### 6.4.10 Function Block – RE_TRIG

Esta *FB* é usada para detectar flancos ascendentes de variáveis e pretende substituir a *FB standard* R_TRIG, devido a esta detectar flancos ascendentes errados duma variável quando ela é inicializada a *true*.

Listagem 18: Function Bock RE_TRIG

```
1.  FUNCTION_BLOCK RE_TRIG
2.  VAR_INPUT  CLK: BOOL; END_VAR
3.  VAR_OUTPUT  Q: BOOL; END_VAR
4.  VAR M: BOOL := FALSE; END_VAR
5.
6.  Q := CLK AND NOT M;
7.  M := CLK;
8.  END_FUNCTION_BLOCK
```

### 6.4.11 Function Block – TIMER_OF_TRANSITION

Esta *FB* é reponsável por controlar as transições temporizadas. Tem como variável de entrada `transition`, que representa a transição, e IN. A *FB* irá proceder à actualização do temporizador (incrementando o seu tempo) se a variável IN tiver o valor *true*, ou fazer reset ao temporizador se tiver o valor *false*. A variável de saída `Q` é devolvida com o valor *true* caso o temporizador da transição tiver atingido o tempo desejado, e só tem sentido quando a chamada da *FB* for feita para actualizar um temporizador (IN a *true*). Uma chamada desta *FB* para uma transição que não seja temporizada irá retornar Q como tendo o valor de `IN`, ou seja, uma chamada à *FB* com o intuito de actualizar transições não temporárias irá retornar sempre verdadeiro, pois considera-se o temporizador atingiu o tempo desejado, apesar de não o ter.

Listagem 19: Function Bock `TIMER_OF_TRANSITION`

```
1.  FUNCTION_BLOCK TIMER_OF_TRANSITION
2.  VAR_INPUT
3.      transition : UINT;
4.      IN : BOOL;
5.  END_VAR
6.  VAR_OUTPUT
7.      Q : BOOL;
8.  END_VAR
9.  VAR
10.     TON_3 : TON;
11.     TON_4 : TON;
12.     TON_5 : TON;
13. END_VAR
14.
15. CASE transition OF
16.     1:    Q := IN;
17.     2:    Q := IN;
18.     3:    TON_3(IN := IN, PT := T#10S);
19.           Q := TON_3.Q;
20.     4:    TON_4(IN := IN, PT := T#30S);
21.           Q := TON_4.Q;
22.     5:    TON_5(IN := IN, PT := T#5S);
23.           Q := TON_5.Q;
24.     6:    Q := IN;
25. END_CASE
26. END_FUNCTION_BLOCK
```

### 6.4.12 Function Block – UPDATE_OUTPUTS

A *FB* UPDATE_OUTPUTS é responsável pela actualização das variáveis de saída, sendo por colocá-las a *false* caso o lugar a que estão associadas deixe de ter marcas, ou por colocá-las a *true* caso o lugar, que não tinha marcas, passe a ter.

Listagem 20: Function Bock UPDATE_OUTPUTS

```
1.  FUNCTION_BLOCK UPDATE_OUTPUTS
2.  VAR_INPUT
3.      place : UINT;
4.  END_VAR
5.
6.  CASE place OF
7.      1:
8.          BUFFERENTRADA := PLACES_TEMP[1].Marking > 0;
9.      2:
10.         ROBOTOCUPADO   :=   PLACES_TEMP[2].Marking   >   0   OR
PLACES_TEMP[3].Marking > 0;
11.     3:
12.         ROBOTOCUPADO   :=   PLACES_TEMP[2].Marking   >   0   OR
PLACES_TEMP[3].Marking > 0;
13.     4:
14.         MAQUINAOCUPADA := PLACES_TEMP[4].Marking > 0;
15.     5:
16.         MAQUINALIVRE := PLACES_TEMP[5].Marking > 0;
17.     6:
18.         ROBOTLIVRE := PLACES_TEMP[6].Marking > 0;
19.     7:
20.         BUFFERSAIDA := PLACES_TEMP[7].Marking > 0;
21. ELSE;
22. END_CASE
23. END_FUNCTION_BLOCK
```

## 6.5 Arquitectura do código gerado para GRAFCET

Os elementos das tabelas 4.1 a 4.9 (do Capítulo 4.2) que possuem uma cruz (“X”), na última coluna da tabela, são os.que estão contemplados no WEBGRAF 2. À excepção de macro-etapas, etapas encapsuladoras e *forcing orders* de congelamento, todos os elementos estão contemplados.

O código gerado a partir de um GRAFCET é constituído por um *program*, uma declaração de variáveis globais e 12 *FBs*. O algoritmo de execução do GRAFCET está dividido em 3 fases:

- **Disparo das transições -** na primeira fase, para cada transição é verificado se os seus lugares de entrada estão activos e se a sua receptividade é verdadeira. Em caso afirmativo, é disparada a transição na rede auxiliar e executadas as eventuais acções de transposição que estejam associadas a essa transição.
- **Execução das acções** - na segunda fase são executadas as acções que estejam associadas a etapas activas do grafcet auxiliar, ou que satisfaçam outros condicionalismos impostos pela acção.
- **Actualização do grafcet original –** na terceira fase o grafcet original é actualizado com os valores da rede auxiliar.

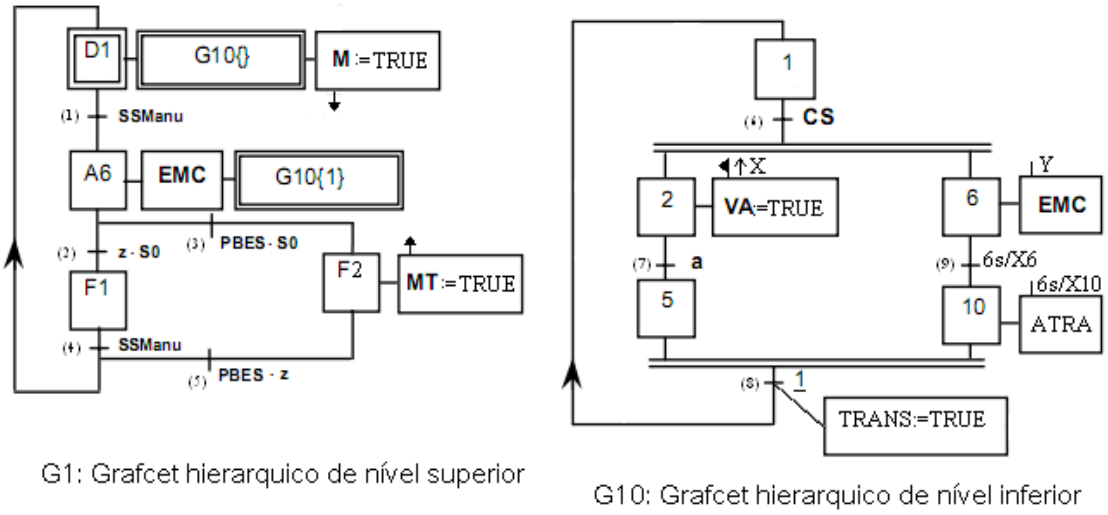
A Tabela 6.2 apresenta a lista das *FBs* geradas com as respectivas variáveis de entrada e saída.

Tabela 6.2: Lista das *Function Blocks* geradas pelo tradutor.

| **Function Block** | **Variáveis de entrada** | **Variáveis de saída** |
|---|---|---|
| CHECK_CONDITION_OF_TRANSITION | TRANSITION : UINT; | CONDITION : BOOL; |
| CHECK_IF_INPUT_STEPS_ARE_ACTIVE | TRANSITION: UINT; | INPUT_STEPS_ACTIVE : BOOL; |
| ENABLE_STEP | STEP:UINT; | |
| DISABLE_STEP | STEP:UINT; | |
| EXECUTE_ACTIONS_OF_TRANSITION | TRANSITION : UINT; | |
| EXECUTE_ACTIONS_ON_ACTIVATION | | |
| EXECUTE_ACTIONS_ON_DEACTIVATION | | |
| EXECUTE_ACTIONS_ON_EVENT | | |
| EXECUTE_CONTINUOUS_ACTIONS | | |
| EXECUTE_FORCING_ORDERS | | |
| IS_STEP_ACTIVE | STEP:UINT; | ACTIVE:BOOL; |
| FIRE_TRANSITION | TRANSITION : UINT; | |
| | | |

À semelhança do código gerado para as redes de Petri, a variável de entrada TRANSITION, partilhada por algumas *FBs*, é uma variável do tipo UINT e é uma referência para uma transição. Em que os valores 1, 2, 3, … n, correspondem à primeira, segunda, terceira, n-ésima transição que aparece no documento GML. A diferença é que as transições não estão guardadas em *arrays* globais, pois não houve necessidade de criar uma estrutura de dados para representar as transições. O mesmo se passa para as etapas.

As secções que se seguem explicam o funcionamente de cada unidade de código gerado pelo tradutor e apresenta um excerto do código relevante. O código gerado será respeitante ao grafcet hierárquico ilustrado na Figura 6.8




G1: Grafcet hierarquico de nível superior

G10: Grafcet hierarquico de nível inferior

Figura 6.9: Grafcet hierárquico.

### 6.5.1 VAR_GLOBAL

De um modo geral, a declaração das variáveis globais incluí:

- Variáveis das etapas;
- Variáveis das etapas do grafcet auxiliar;
- Variáveis de entrada;
- Variáveis de saída .

O código apresentado na Listagem 21 a seguir corresponde à declaração de variáveis globais gerada pelo tradutor relativo ao GRAFCET da Figura 6.9.

Listagem 21: Código da declaração da VAR_GLOBAL

```
1. VAR_GLOBAL
2.
3.  (* STEPS *)
4. XD1 : BOOL := TRUE;
5. XA6 : BOOL;
6. XF1 : BOOL;
7. XF2 : BOOL;
8. X1 : BOOL;
9. X2 : BOOL;
10.X5 : BOOL;
11.X6 : BOOL;
12.X10 : BOOL;
13.
14.(* VIRTUAL STEPS *)
15.XD1_TEMP : BOOL := TRUE;
16.XA6_TEMP : BOOL;
17.XF1_TEMP : BOOL;
```

```
18. XF2_TEMP : BOOL;
19. X1_TEMP : BOOL;
20. X2_TEMP : BOOL;
21. X5_TEMP : BOOL;
22. X6_TEMP : BOOL;
23. X10_TEMP : BOOL;
24.
25. TRANSITION_LENGTH : UINT := 9;
26.
27. (* VARIABLES *)
28. SSManu : BOOL;
29. TRANS : BOOL;
30. Z : BOOL;
31. S0 : BOOL;
32. PBES : BOOL;
33. X : BOOL;
34. Y : BOOL;
35. A : BOOL;
36. CS : BOOL;
37. EMC : BOOL;
38. MT : BOOL;
39. ATRA : BOOL;
40. M : BOOL;
41. VA : BOOL;
42.
43. END_VAR
```

`XD1`, `XA6`, `XF1`, …, `X10` correspondem às *step variables* das etapas `D1`, `A6`, `F1`, …, `10`. Possuem o valor *true* caso a etapa correspondente esteja activa, ou *false* no caso contrário. As *step variables* das etapas iniciais são inicializadas a *true*. `XD1_TEMP`, `XA6_TEMP`, `XF1_TEMP`, …, `X10_TEMP` representam as *step variables* da etapas do GRAFCET auxiliar.

`TRANSITION_LENGTH` é uma variável que contém o número total de transições existentes no GRAFCET.

As restantes variáveis `SSManu`, `TRANS`, …, `VA`, correspondem às variáveis presentes nas acções e condições das transições. Estas podem ser tanto variáveis de saída (no caso de estarem associadas a acções) como de entrada (associadas a condições de transições).

### 6.5.2 PROGRAM – MAIN_PROGRAM

Seguidamente é apresentado o corpo do *program.*

Listagem 22: Código respeitante ao do *program.*

```
1.  IF FIRSTCYCLE THEN
2.          EXECUTE_FORCING_ORDERS();
3.          EXECUTE_CONTINUOUS_ACTIONS();
```

```
4.        FIRSTCYCLE := FALSE;
5.  END_IF
6.
7.  (* for each transition *)
8.  FOR TRANSITION := 1 TO TRANSITION_LENGTH DO
9.        CHECK_IF_INPUT_STEPS_ARE_ACTIVE(TRANSITION          :=
    TRANSITION, INPUT_STEPS_ACTIVE => INPUT_STEPS_ACTIVE);
10.       CHECK_CONDITION_OF_TRANSITION(TRANSITION := TRANSITION,
    CONDITION => CONDITION);
11.       IF INPUT_STEPS_ACTIVE AND CONDITION THEN
12.             FIRE_TRANSITION(TRANSITION := TRANSITION);
13.             EXECUTE_ACTIONS_OF_TRANSITION(TRANSITION       :=
    TRANSITION);
14.       END_IF
15. END_FOR
16.
17. (* EXECUTE ACTIONS *)
18. EXECUTE_CONTINUOUS_ACTIONS();
19. EXECUTE_ACTIONS_ON_EVENT();
20. EXECUTE_ACTIONS_ON_ACTIVATION();
21. EXECUTE_ACTIONS_ON_DEACTIVATION();
22. EXECUTE_FORCING_ORDERS();
23.
24. (* update net *)
25. XD1:=XD1_TEMP;
26. XA6:=XA6_TEMP;
27. XF1:=XF1_TEMP;
28. XF2:=XF2_TEMP;
29. X1:=X1_TEMP;
30. X2:=X2_TEMP;
31. X6:=X6_TEMP;
32. X5:=X5_TEMP;
33. X10:=X10_TEMP;
```

Como já referido anteriormente, o algoritmo de execução do programa do sistema de controlo está dividido em 3 fases. A primeira procede ao disparo das transições (criando a representação do novo estado na rede auxiliar). Através da análise ao código pode-se ver que o ciclo FOR percorre todas as transições, e para cada uma delas verifica se os seus lugares de entrada estão activos (*FB* CHECK_IF_INPUT_STEPS_ARE_ACTIVE) e se a sua receptividade é verdadeira (*FB* CHECK_CONDITION_OF_TRANSITION), se estas duas condições forem satisfeitas procede-se ao disparo da transição (*FB* FIRE_TRANSITION) e à execução de eventuais acções de transposição que possa ter (*FB* EXECUTE_ACTIONS_OF_TRANSITION).

Na segunda fase procede-se à execução de todas as acções e *forcing orders*, que é realizado pelas seguintes *FBs:*

- EXECUTE_CONTINUOUS_ACTIONS;

- EXECUTE_ACTIONS_ON_EVENT;
- EXECUTE_ACTIONS_ON_ACTIVATION;
- EXECUTE_ACTIONS_ON_DEACTIVATION;
- EXECUTE_FORCING_ORDERS.

A terceira fase procede à actualização da rede original com os valores da rede auxiliar, que corresponde à úlima parte do código apresentado.

Todas as *FBs* apresentadas no código irão ser apresentadas nas sub-secções seguintes.

### 6.5.3 Function Block – CHECK_CONDITION_OF_TRANSITION

Esta *FB* verifica se a receptividade de uma dada transição(`TRANSITION`) é verdadeira ou falsa, colocando esse valor na variável de saída `CONDITION`.

Listagem 23: Código da *FB* CHECK_CONDITION_OF_TRANSITION

```
1.    FUNCTION_BLOCK CHECK_CONDITION_OF_TRANSITION
2.    VAR_INPUT
3.       TRANSITION : UINT;
4.    END_VAR
5.    VAR_OUTPUT
6.       CONDITION : BOOL;
7.    END_VAR
8.    VAR
9.       TON_9 : TON;
10.   END_VAR
11.   CASE TRANSITION OF
12.      1:
13.            CONDITION := SSManu;
14.      2:
15.            CONDITION := Z AND S0;
16.      3:
17.            CONDITION := PBES AND S0;
18.      4:
19.            CONDITION := SSManu;
20.      5:
21.            CONDITION := PBES AND Z;
22.      6:
23.            CONDITION := CS;
24.      7:
25.            CONDITION := A;
26.      8:
27.            CONDITION := TRUE;
28.      9:
29.            TON_9(IN := X6, PT := T#6s);
30.            CONDITION := TON_9.Q;
31.            IF CONDITION THEN
32.                  TON_9(IN := FALSE);
```

```
33.            END_IF
34.      ELSE
35.            CONDITION := FALSE;
36.   END_CASE
37.   END_FUNCTION_BLOCK
```

### 6.5.4 Function Block – CHECK_IF_INPUT_STEPS_ARE_ACTIVE

Esta *FB* verifica se todos os lugares de entrada de uma dada transição (`TRANSITION`), estão activos.

Listagem 24: Código da *FB* CHECK_IF_INPUT_STEPS_ARE_ACTIVE

```
1. FUNCTION_BLOCK CHECK_IF_INPUT_STEPS_ARE_ACTIVE
2. VAR_INPUT
3.       TRANSITION: UINT;
4. END_VAR
5. VAR_OUTPUT
6.       INPUT_STEPS_ACTIVE: BOOL;
7. END_VAR
8. VAR
9. END_VAR
10.      CASE TRANSITION OF
11.            1:
12.                  INPUT_STEPS_ACTIVE := XD1;
13.            2:
14.                  INPUT_STEPS_ACTIVE := XA6;
15.            3:
16.                  INPUT_STEPS_ACTIVE := XA6;
17.            4:
18.                  INPUT_STEPS_ACTIVE := XF1;
19.            5:
20.                  INPUT_STEPS_ACTIVE := XF2;
21.            6:
22.                  INPUT_STEPS_ACTIVE := X1;
23.            7:
24.                  INPUT_STEPS_ACTIVE := X2;
25.            8:
26.                  INPUT_STEPS_ACTIVE := X5 AND X10;
27.            9:
28.                  INPUT_STEPS_ACTIVE := X6;
29.            ELSE
30.                 INPUT_STEPS_ACTIVE := FALSE;
31.      END_CASE
32.      END_FUNCTION_BLOCK
```

### 6.5.5 Function Block – ENABLE_STEP

Esta *FB* activa um dado lugar. As alterações são feitas à rede auxiliar. Vale a pena relembrar que no método síncrono as alterações à rede são feitas na rede auxiliar, e só no final de cada ciclo é que a rede original é actualizada. A activação de uma etapa corresponde à atribuição do valor *true* à *step variable* correspondente.

Listagem 25: Código da *FB* ENABLE_STEP

```
1. FUNCTION_BLOCK ENABLE_STEP
2. VAR_INPUT
3.       STEP:UINT;
4. END_VAR
5. VAR_OUTPUT
6. END_VAR
7. VAR
8. END_VAR
9. CASE STEP OF
10.            1:
11.                  XD1_TEMP:=TRUE;
12.            2:
13.                  XA6_TEMP:=TRUE;
14.            3:
15.                  XF1_TEMP:=TRUE;
16.            4:
17.                  XF2_TEMP:=TRUE;
18.            5:
19.                  X1_TEMP:=TRUE;
20.            6:
21.                  X2_TEMP:=TRUE;
22.            7:
23.                  X5_TEMP:=TRUE;
24.            8:
25.                  X6_TEMP:=TRUE;
26.            9:
27.                  X10_TEMP:=TRUE;
28.            ELSE;
29.      END_FUNCTION_BLOCK
```

### 6.5.6 Function Block – DISABLE_STEP

Esta *FB* desactiva um dado lugar. As alterações são, uma vez mais, feitas à rede auxiliar. A desactivação de uma etapa corresponde à atribuição do valor *false* à *step variable* dessa etapa.

Listagem 26: Código da *FB* DISABLE_STEP

```
1.  FUNCTION_BLOCK DISABLE_STEP
2.  VAR_INPUT
```

```
3.       STEP:UINT;
4.  END_VAR
5.
6.  CASE STEP OF
7.       1:
8.            XD1_TEMP:=FALSE;
9.       2:
10.           XA6_TEMP:=FALSE;
11.      3:
12.           XF1_TEMP:=FALSE;
13.      4:
14.           XF2_TEMP:=FALSE;
15.      5:
16.           X1_TEMP:=FALSE;
17.      6:
18.           X2_TEMP:=FALSE;
19.      7:
20.           X5_TEMP:=FALSE;
21.      8:
22.           X6_TEMP:=FALSE;
23.      9:
24.           X10_TEMP:=FALSE;
25.      ELSE;
26. END_CASE
27. END_FUNCTION_BLOCK
```

### 6.5.7 Function Block – EXECUTE_ACTIONS_OF_TRANSITION

Esta *FB* executa a acção de transposição associada a uma dada transição, `TRANSITION`.

Listagem 27: Código da *FB* EXECUTE_ACTIONS_OF_TRANSITION

```
1.  FUNCTION_BLOCK EXECUTE_ACTIONS_OF_TRANSITION
2.  VAR_INPUT
3.       TRANSITION : UINT;
4.  END_VAR
5.
6.  CASE TRANSITION OF
7.       8:
8.            TRANS:=TRUE;
9.       ELSE;
10. END_CASE
11. END_FUNCTION_BLOCK
```

### 6.5.8 Function Block – EXECUTE_ACTIONS_ON_ACTIVATION

Esta *FB* executa todas as acções de activação cujas etapas a que estão associadas acabaram de ficar activas.

Listagem 28: Código da *FB* EXECUTE_ACTIONS_ON_ACTIVATION

```
1.  FUNCTION_BLOCK EXECUTE_ACTIONS_ON_ACTIVATION
2.  VAR
3.        RE_XF2 : RE_TRIG;
4.  END_VAR
5.
6.  RE_XF2(CLK := XF2);
7.  IF RE_XF2.Q THEN
8.        MT := TRUE;
9.  END_IF
10. END_FUNCTION_BLOCK
```

Através da análise ao código, pode-se reparar que a *FB* `RE_TRIG` é usada para detectar um *rising edge* da *step variable* `XF2`. E se for o caso, `MT := TRUE` é realizado.

### 6.5.9 Function Block – EXECUTE_ACTIONS_ON_DEACTIVATION

Esta *FB* executa todas as acções de activação cujas etapas a que estão associadas acabaram de ficar desactivas.

Listagem 29: Código da *FB* EXECUTE_ACTIONS_ON_DEACTIVATION

```
1.  FUNCTION_BLOCK EXECUTE_ACTIONS_ON_DEACTIVATION
2.  VAR
3.        F_XD1 : F_TRIG;
4.  END_VAR
5.
6.  F_XD1(CLK := XD1);
7.  IF F_XD1.Q THEN
8.        M := TRUE;
9.  END_IF
10. END_FUNCTION_BLOCK
```

A *FB* standard *F*_TRIG é usada para detectar um *falling edge* da *step variable* XD1. E se for o caso, M := TRUE é realizado.

### 6.5.10 Function Block – EXECUTE_ACTIONS_ON_EVENT

Esta *FB* é responsável pela execução de todas as acções de evento. Uma acção de evento pode executar se a etapa a que está associada estiver activa e se se verificar a condição da acção.

Listagem 30: Código da *FB* EXECUTE_ACTIONS_ON_EVENT

```
1.  FUNCTION_BLOCK EXECUTE_ACTIONS_ON_EVENT
2.  VAR
3.        RE_X_1 : RE_TRIG;
4.  END_VAR
5.
6.  RE_X_1(CLK := X);
7.  IF X2 AND (RE_X_1.Q) THEN
8.        VA := TRUE;
9.  END_IF
10. END_FUNCTION_BLOCK
```

Neste caso, o estado da etapa é dado pela *step variable* `X2` e a condição ("↑X") é dada por `RE_X_1.Q`. `RE_X_1` é uma instância da *FB* `RE_TRIG.[MOR00]`

### 6.5.11 Function Block – EXECUTE_CONTINUOUS_ACTIONS

Esta *FB* é responsável pela execução das acções contínuas.

Listagem 31: Código da *FB* EXECUTE_CONTINUOUS_ACTIONS

```
1.  FUNCTION_BLOCK EXECUTE_CONTINUOUS_ACTIONS
2.  VAR
3.        TON_2 : TON;
4.  END_VAR
5.
6.  IF X10 THEN
7.        TON_2(IN := X10, PT := T#6s);
8.  ELSE
9.        TON_2(IN := FALSE);
10. END_IF
11. ATRA := TON_2.Q;
12. EMC := XA6 OR X6 AND Y;
13. END_FUNCTION_BLOCK
```

No exemplo da Figura 6.9 as acções contínuas estão associadas às etapas A6, 6 e 10.

### 6.5.12 Function Block – EXECUTE_FORCING_ORDERS

Esta *FB* é responsável pela execução das *forcing orders*, que corresponde à colocação de um grafcet parcial numa dada situação, ou seja, activar ou desactivar as etapas dum grafcet parcial consoante a ordem da *forcing order*.

Listagem 32: Código da *FB* EXECUTE_FORCING_ORDERS

```
1.  FUNCTION_BLOCK EXECUTE_FORCING_ORDERS
```

```
2.  IF XD1 THEN
3.        X1_TEMP := FALSE;
4.        X2_TEMP := FALSE;
5.        X5_TEMP := FALSE;
6.        X6_TEMP := FALSE;
7.        X10_TEMP := FALSE;
8.  END_IF
9.  IF XA6 THEN
10.       X1_TEMP := TRUE;
11.       X2_TEMP :=FALSE;
12.       X5_TEMP :=FALSE;
13.       X6_TEMP :=FALSE;
14.       X10_TEMP :=FALSE;
15. END_IF
16. END_FUNCTION_BLOCK
```

A forcing order associada à etapa D1 desactiva todas as etapas do grafcet parcial G10, ou seja, coloca as *step variables* das etapas de G10 a *false.* A *forcing order* associada à etapa A6 coloca G10 na situação G10{1}, ou seja, apenas a etapa 1 fica activa (`X1_TEMP`), todas as outras ficam inactivas.

Na definição do *program* (sub-secção 6.5.2) a *FB* `EXECUTE_FORCING_ORDERS` é a última a ser chamada, garantindo que possíveis mudanças a um grafcet forçado não sejam visíveis ao PLC, pois irão voltar a ter os valores correctos antes das variáveis de saída do PLC serem actualizadas, que acontece no final de cada ciclo.

### 6.5.13 Function Block – IS_STEP_ACTIVE

Esta FB indica se uma dada etapa , `STEP`, está activa, colocando o estado da etapa na variável de saída `ACTIVE`. Essa informação é dada pelas *step variables* das etapas., estando uma etapa activa ou não consoante a respectiva *step variable* esteja a *true* ou *false*;

```
1.  FUNCTION_BLOCK IS_STEP_ACTIVE
2.  VAR_INPUT
3.        STEP:UINT;
4.  END_VAR
5.  VAR_OUTPUT
6.        ACTIVE:BOOL;
7.  END_VAR
8.  VAR
9.  END_VAR
10.
11. CASE STEP OF
12.       1:
13.             ACTIVE := XD1;
14.       2:
```

```
15.             ACTIVE := XA6;
16.       3:
17.             ACTIVE := XF1;
18.       4:
19.             ACTIVE := XF2;
20.       5:
21.             ACTIVE := X1;
22.       6:
23.             ACTIVE := X2;
24.       7:
25.             ACTIVE := X5;
26.       8:
27.             ACTIVE := X6;
28.       9:
29.             ACTIVE := X10;
30.       ELSE;
31. END_CASE
32. END_FUNCTION_BLOCK
```

### 6.5.14 Function Block – FIRE_TRANSITION

Esta *FB* dispara uma dada transição, desactivando os seus lugares de entrada e activando os lugares de saída.

```
1.  FUNCTION_BLOCK FIRE_TRANSITION
2.  VAR_INPUT
3.        TRANSITION : UINT;
4.  END_VAR
5.  VAR
6.        ENABLE_STEP : ENABLE_STEP;
7.        DISABLE_STEP : DISABLE_STEP;
8.  END_VAR
9.  CASE TRANSITION OF
10.       1:
11.             DISABLE_STEP(STEP := 1);
12.             ENABLE_STEP(STEP := 2);
13.       2:
14.             DISABLE_STEP(STEP := 2);
15.             ENABLE_STEP(STEP := 3);
16.       3:
17.             DISABLE_STEP(STEP := 2);
18.             ENABLE_STEP(STEP := 4);
19.       4:
20.             DISABLE_STEP(STEP := 3);
21.             ENABLE_STEP(STEP := 1);
```

```
22.       5:
23.             DISABLE_STEP(STEP := 4);
24.             ENABLE_STEP(STEP := 1);
25.       6:
26.             DISABLE_STEP(STEP := 5);
27.             ENABLE_STEP(STEP := 6);
28.             ENABLE_STEP(STEP := 8);
29.       7:
30.             DISABLE_STEP(STEP := 6);
31.             ENABLE_STEP(STEP := 7);
32.       8:
33.             DISABLE_STEP(STEP := 7);
34.             DISABLE_STEP(STEP := 9);
35.             ENABLE_STEP(STEP := 5);
36.       9:
37.             DISABLE_STEP(STEP := 8);
38.             ENABLE_STEP(STEP := 9);
39. END_CASE
40. END_FUNCTION_BLOCK
```

### 6.5.15 Function Block - RE_TRIG

A FB RE_TRIG é equivalente à gerada para as redes de Petri (ver sub-secção 6.4.10 ).

## 6.6 Exemplo de aplicação

Este capítulo demonstra a utilização da ferramenta WEBGRAF 2 tanto em modo Cliente -Servidor como em modo Standalone.

### 6.6.1 Modo Servidor-Cliente

A utilização da aplicação WEBGRAF 2 é muito simples. Basicamente, após se ter o site da aplicação aberto, basta seleccionar o ficheiro (GML ou PNML) que se pretende traduzir, escolher a linguagem de programação alvo, e submeter o ficheiro (ver Figura 6.10). Imediatamente a seguir é pedido ao utilizador que proceda ao download do ficheiro gerado. Este poderá posteriormente ser passado para a ferramenta de desenvolvimento do utilizador por via de *copy-paste.*

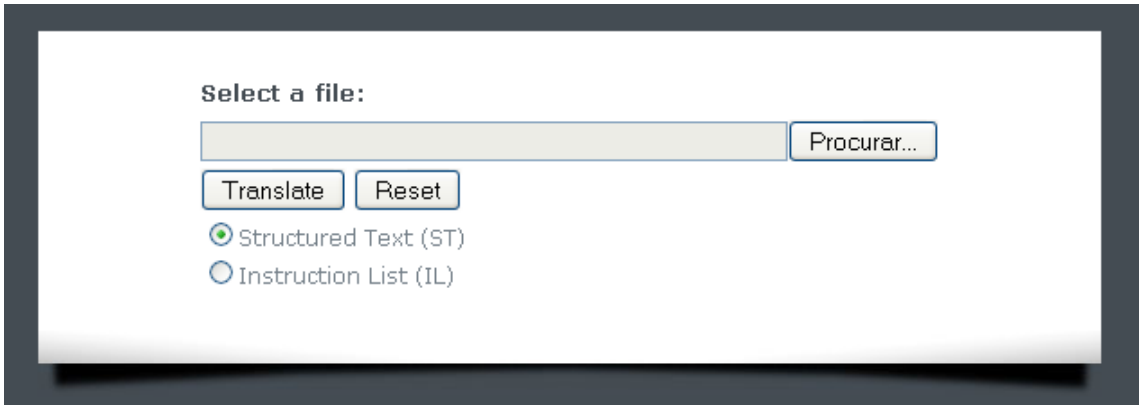



Figura 6.10: Interface que permite a selecção e envio do ficheiro contendo a explicitação textual do modelo de controlo sequencial para o servidor.

### 6.6.2 Modo Standalone

A versão standalone consiste num ficheiro JAR executável que pode ser executando fazendo um duplo clique sobre ele. Nesse caso, é esperado que exista um ficheiro com nome "default.pnml" na mesma directoria do JAR.


Figura 6.11: Exemplo 1 para ilustrar a utilização da aplicação WEBGRAF 2.

O ficheiro traduzido será criado com o nome "default.txt".


Figura 6.12: Exemplo 2 para ilustrar a utilização da aplicação WEBGRAF 2.

O ficheiro JAR também pode ser executado via linha de comandos, sendo o comando de invocação geral: **java -jar webgraf2.jar [-f filePath] [-s savePath] [-il]**. A inclusão do argumento -help apresenta o menu de ajuda.


Figura 6.13: Exemplo de invocação da aplicação WEBGRAF 2 pela linha de comandos.

Os argumentos da linha de comandos são:

- **[-f filePath]** – path do ficheiro a ser convertido. Em caso de omissão deste argumento, é convertido o ficheiro com nome "default.pnml";
- **[-s savePath]** – guarda o ficheiro gerado na localização especificada. Em caso de omissão será guardado com o mesmo nome do ficheiro de entrada com extensão ".txt";
- **[-il]** – faz com que o código gerado esteja escrito em IL. Em caso de omissão deste argumento o programa é escrito em ST.
- **[-help]** – este argumento faz aparecer o menu de ajuda.

# 7 Conclusões e Trabalho Futuro

O trabalho realizado teve como objectivo o desenvolvimento de uma ferramenta de tradução de modelos de controlo sequencial em código de programação normalizado dos PLCs. Mais concretamente, pretende-se com esta ferramenta, de nome WEBGRAF 2, traduzir sistemas modelados em redes de Petri do tipo lugar/transição (com algumas extensões) ou GRAFCETs. A tradução é feita a partir duma explicitação textual dos modelos e, portanto, o Capítulo 2, destinado à revisão bibliográfica, introduz um formato normalizado (baseado em XML) para representação de redes de Petri. O objectivo do uso de um formato normalizado prende-se com o facto de permitir que a rede possa ser transferida para outras ferramentas que permitam fazer análise e simulação das redes. No entanto, não existe nenhum formato normalizado para explicitação de GRAFCETs, e surgiu portanto a necessidade de se criar um formato próprio para a aplicação, mas que aproveitasse o máximo possível eventuais formatos existentes. O Capítulo 2 apresenta então o PLCOpen XML, que surgiu da necessidade da procura por formatos existentes de explicitação de GRAFCETs. Este formato contém a especificação da linguagem estrutural SFC que é uma linguagem de PLCs que surgiu a partir do GRAFCET. Apesar de ser diferente do GRAFCET, possui vários elementos em comum. Foi criado então o GML, uma linguagem de especificação de GRAFCETs baseada em XML, que possui várias semelhanças com a linguagem XML de especificação de SFCs. Ainda relativamente ao Capítulo 2, procedeu-se à procura de ferramentas que permitissem o desenho dos modelos de controlo sequencial usados pela ferramenta, e exportação para os seus respectivos formatos XML. Daí surgiu a ferramenta de nome JARP que permite o desenho e simulação de redes de Petri e exportação para o formato PNML. Esta ferramenta é um auxiliar fundamental da ferramenta WEBGRAF 2 na medida em que permite uma rápida geração do ficheiro PNML a partir do desenho de uma rede de Petri. Ferramentas para geração de GML, logicamente, não existem.

O Capítulo 3 destinou-se a apresentar umas noções básicas sobre redes de Petri e também o correspondente formato XML para especificação das redes – PNML.

Já o Capítulo 4 fez uma apresentação da linguagem GRAFCET e do correspondente formato de especificação proposto – GML.

Visto que a ferramenta WEBGRAF 2 gera programas de controlo sequencial escritos em linguagens de PLCs, há a necessidade de apresentar a estrutura básica de um PLC e explicar o seu funcionamento geral, que é feito no Capítulo 5. Esse mesmo capítulo também é responsável por fazer uma breve introdução às linguagens de programação de PLCs geradas pela aplicação WEBGRAF 2. Essas linguagens são IL e ST.

No Capítulo 6 foram apresentadas: a arquitectura geral da aplicação e as tecnologias utilizadas no seu desenvolvimento; a arquitectura geral do código gerado pela aplicação, e discutidas as várias unidades de código geradas; e foi dado um exemplo de utilização da aplicação.

Finalmente, o presente capítulo, capítulo 7, destina-se a presentar as conclusões do trabalho e perspectivas de prossecução.

Acredita-se que esta ferramenta possa funcionar como ferramenta de e-learning na área dos sistemas de controlo lógico. O facto de ser uma ferramenta Web, torna-a acessível a toda a gente, e faz dela uma ferramenta inovadora, visto não existirem ferramentas com estas duas características simultaneamente: ser uma aplicação WEB e gerar programas escritos em código normalizado de PLCs.

## 7.1 Satisfação dos Objectivos

O tradutor abrange por completo as redes de Petri do tipo Lugar/Transição e algumas extensões (transições temporizadas). No caso do GRAFCET abrange todos os tipos de etapas, acções, transições e *forcing orders* com excepção das macro-acções, etapas encapsuladoras (*enclosing step*). Os modelos de controlo sequencial são correctamente traduzidos para ambas as linguagens previstas: ST e IL. As principais funcionalidades da aplicação foram todas implementadas, e esta encontra-se funcional e disponível na Web.

Os pontos de melhoria em relação à versão anterior do WEBGRAF são: utilização de um formato normalizado para especificação das redes de Petri – este ponto trás claras vantagens em relação à primeira versão do WEBGRAF, pois permite a criação facilitada do modelo com editores gráficos existentes, e permite ainda a transferência do modelo para outras ferramentas de análise que utilizem o formato normalizado; geração de código PLC normalizado – outro ponto que trás melhorias significativas, pois desta forma o código gerado não tem um determinado PLC como alvo, permitindo que o código seja compatível com qualquer PLC existente ou qualquer PLC que venha a existir; e, finalmente, geração de código ST. Ao contrário da versão anterior do WEBGRAF, que só gera programas IL, o WEBGRAF 2 suporta geração tanto para IL como para ST. Por ser uma linguagem parecida com “C” ou PASCAL, permite que os programadores familiarizados com estas linguagens se adaptem facilmente a ela.

## 7.2 Trabalho Futuro

Como trabalho futuro salienta-se a importância de se desenvolver um editor gráfico de GRAFCETs com possibilidade de exportação para o formato GML. O suporte a mais extensões de redes de Petri, como por exemplo: redes de Petri coloridas, também constitui um ponto de elevado interesse. Para finalizar, falta ao tradutor contemplar macro-acções e lugares encapsuladores (*enclosing steps*).

# Referências


[BCHK03] Jonathan Billington, Soren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber, The Petri Net Markup Language: Concepts, Technology, and Tools. March 2003.

[Bol09] Bolton, W. Programmable Logic Controllers, 5th edition. Newnes, 2009.

[CL99] Cassandras, C. and Lafortune, S. Introduction to discrete event systems - Kluwer Academic Publishers, Boston, 1999.

[Coe05] Coelho, P. *Programação com JSP - Curso completo*. FCA, Lisbon, 2005.

[DOM00] Document Object Model (DOM). http://www.w3.org/DOM/

[Gom03] Gomes, A. *WEBGRAF - Aplicação Web para execução de GRAFCETs e Redes de Petri em Controladores Lógicos Programáveis*. Dissertation MAIC, FEUP 2003.

[GM10] Garcês, R., Magalhães, A. A Webtool for generating standard PLC code from petri nets specifications, to appear in proceedings of Controlo2010, Coimbra, Portugal, September 2010.

[Har02] Harold, E. *Processing XML with Java*, Addison-Wesley Professional, 2002.

[Har01] Harold, E. *XML Bible (2nd Edition)*. Wiley, 2001.

[Mur89] Murata, T. Petri nets: Properties, analysis and applications. Proc. IEEE Vol 77 (4), pp 541–580, 1989.

[IEC02] IEC 60848 INT. STANDARD, 2nd Ed. GRAFCET specification language for sequential function charts. Reference number CEI/IEC 60848, 2002.

[IEC93] IEC 61131-3 INT. STANDARD. Programmable controllers - Part 3: Programming languages, 1993.

[ISO09] ISO/IEC 15909-2 INT. STANDARD. Software and system engineering - High-level Petri nets - Part 2: Transfer Format, 2009.

[ISO04] ISO/IEC 15909-1 INT. STANDARD. Systems and software engineering - High-level Petri nets - Part 1: Concepts, definitions and graphical notation, 2004.

[JARP01] JARP Project - Petri Nets Analyzer, 2001. http://jarp.sourceforge.net/

[Kin06] Ekkart Kindler Paper for the invited talk at EKA 2006: In E. Schnieder (ed.): Entwurf Komplexer Automatisierungssysteme, EKA 2006, 9. Fachtagung, Braunschweig, Germany, pp. 35-55, May 2006.

[PLLD09] PetriLLd – Homepage. http://sourceforge.net/apps/mediawiki/petrilld/index.php?title=Main_Page

[PLCO03] PLCOpen Home. http://www.plcopen.org/

[PNML09] Pnml.org - PNML reference site, 2009. http://www.pnml.org/

[POX09] Technical Paper PLCopen Technical Committee 6 - XML Formats for IEC 61131-3, Version 2.01 – Official Release, May 2009.

[UML10] Object Management Group – UML, http://www.uml.org/

[Web10] WEBGRAF 2 – Homepage. http://yoda.fe.up.pt:8080/ei05068/[1]

[FM00] G. Frey, M. Minas, Editing, Visualizing, and Implementing Signal Interpreted Petri nets, Em Proceedings of the AWPN 2000, Koblenz 2000.

[SIPN02] SIPN Editor – Programming and Visualizing PLC programs with Signal Interpreter Petri Nets. http://www.eit.uni-kl.de/litz/ENGLISH/software/SIPNEditor.htm

1 Apenas acessível dentro da rede da Faculdade de Engenharia da Universidade do Porto (FEUP), ou por ligação VPN.

# Anexo A: Tradução de uma rede de Petri especificada em PNML para código ST

O presente anexo, apresenta uma rede de Petri especificada em PNML e o correspondente código PLC gerado pela aplicação WEBGRAF 2.

## A.1 Rede de Petri de ilustração

A figura seguinte ilustra a rede de Petri a servir de exemplo:

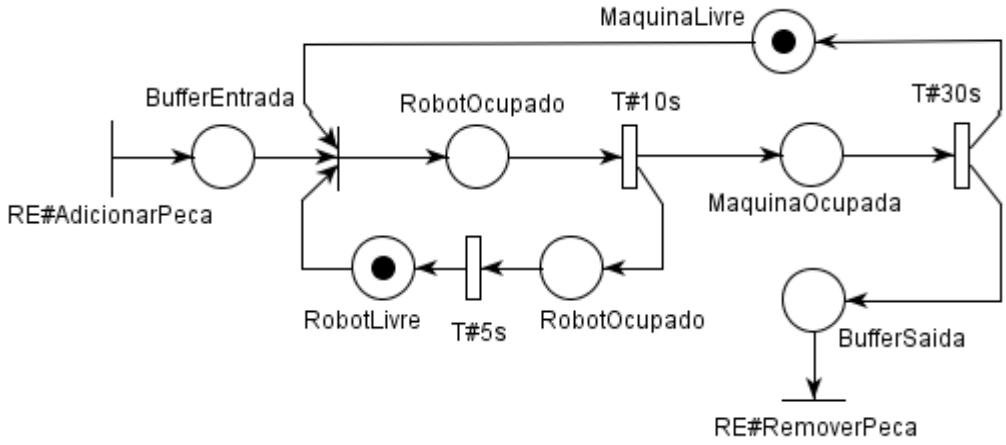



Figura A.1: Rede de Petri exemplo.

## A.2 Especificação do modelo em PNML

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="n1"
  type="http://www.pnml.org/version-2009/grammar/ptnet">
    <page id="page1">
      <name>
        <text>P/T net</text>
      </name>

    <!-- places -->
      <place id="p1">
        <name>
          <text>BufferEntrada</text>
        </name>
        <initialMarking>
          <text>0</text>
        </initialMarking>
        <toolspecific tool="JARP" version="1.2">
      </place>
      <place id="p2">
        <name>
          <text>RobotOcupado</text>
        </name>
        <initialMarking>
          <text>0</text>
        </initialMarking>
      </place>
      <place id="p3">
        <name>
          <text>MaquinaOcupada</text>
        </name>
        <initialMarking>
          <text>0</text>
        </initialMarking>
      </place>
      <place id="p4">
        <name>
          <text>RobotOcupado</text>
        </name>
        <initialMarking>
          <text>0</text>
        </initialMarking>
      </place>
      <place id="p5">
```

```
      <name>
        <text>RobotLivre</text>
      </name>
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <place id="p6">
      <name>
        <text>MaquinaLivre</text>
      </name>
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <place id="p7">
      <name>
        <text>BufferSaida</text>
      </name>
      <initialMarking>
        <text>0</text>
      </initialMarking>
    </place>

  <!-- transitions -->
    <transition id="t1">
      <name>
        <text>RE#AdicionarPeca</text>
      </name>
    </transition>
    <transition id="t2">
      <name>
        <text></text>
      </name>
    </transition>
    <transition id="t3">
      <name>
        <text>T#10s</text>
      </name>
    </transition>
    <transition id="t4">
      <name>
        <text>T#5s</text>
      </name>
    </transition>
    <transition id="t5">
      <name>
```

```
        <text>T#30s</text>
      </name>
    </transition>
    <transition id="t6">
      <name>
        <text>RE#RemoverPeca</text>
      </name>
    </transition>

  <!-- arcs -->
    <arc id="a1" source="t1" target="p1">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a2" source="t4" target="p5">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a3" source="p4" target="t4">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a4" source="t3" target="p3">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a5" source="t3" target="p4">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a6" source="p2" target="t3">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a7" source="t2" target="p2">
      <inscription>
        <text>1</text>
      </inscription>
    </arc>
    <arc id="a8" source="p1" target="t2">
      <inscription>
```

```
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a9" source="p5" target="t2">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a10" source="p6" target="t2">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a11" source="t5" target="p6">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a12" source="t5" target="p7">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a13" source="p7" target="t6">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
    </page>
  </net>
</pnml>
```

## A.3 Código ST gerado a partir do modelo em PNML

```
TYPE
    (* Structure of places *)
    PLACE : STRUCT
        Marking : UINT;
    END_STRUCT
END_TYPE


TYPE
    (* Structure of transitions *)
    TRANSITION : STRUCT
```

```
        Source        : ARRAY [1..3] OF UINT;
        Target        : ARRAY [1..2] OF UINT;
        Source_weight : ARRAY [1..3] OF UINT;
        Target_weight : ARRAY [1..2] OF UINT;
        Source_len    : UINT;
        Target_len    : UINT;
    END_STRUCT
END_TYPE


VAR_GLOBAL (* Global variables *)

    PLACES      : ARRAY [1..7] OF PLACE := [
    (* PLACE 1 *)
    (
        Marking:=0
    ),
    (* PLACE 2 *)
    (
        Marking:=0
    ),
    (* PLACE 3 *)
    (
        Marking:=0
    ),
    (* PLACE 4 *)
    (
        Marking:=0
    ),
    (* PLACE 5 *)
    (
        Marking:=1
    ),
    (* PLACE 6 *)
    (
        Marking:=1
    ),
    (* PLACE 7 *)
    (
        Marking:=0
    )];

    TRANSITIONS : ARRAY [1..6] OF TRANSITION := [
    (* TRANSITION 1 *)
    (
        Source          := [0,0,0],
        Target          := [1,0],
```

```
    Source_weight   := [0,0,0],
    Target_weight   := [1,0],
    Source_len      := 0,
    Target_len      := 1
),
(* TRANSITION 2 *)
(
    Source          := [1,5,6],
    Target          := [2,0],
    Source_weight   := [1,1,1],
    Target_weight   := [1,0],
    Source_len      := 3,
    Target_len      := 1
),
(* TRANSITION 3 *)
(
    Source          := [2,0,0],
    Target          := [3,4],
    Source_weight   := [1,0,0],
    Target_weight   := [1,1],
    Source_len      := 1,
    Target_len      := 2
),
(* TRANSITION 4 *)
(
    Source          := [4,0,0],
    Target          := [5,0],
    Source_weight   := [1,0,0],
    Target_weight   := [1,0],
    Source_len      := 1,
    Target_len      := 1
),
(* TRANSITION 5 *)
(
    Source          := [0,0,0],
    Target          := [6,7],
    Source_weight   := [0,0,0],
    Target_weight   := [1,1],
    Source_len      := 0,
    Target_len      := 2
),
(* TRANSITION 6 *)
(
    Source          := [7,0,0],
    Target          := [0,0],
    Source_weight   := [1,0,0],
    Target_weight   := [0,0],
```

```
        Source_len      := 1,
        Target_len      := 0
    )];

    (* NUMBER OF TRANSITIONS *)
    TRANS_LENGTH : UINT := 6;

    (* OUTPUTS *)
    BufferEntrada : BOOL;
    RobotOcupado : BOOL;
    MaquinaOcupada : BOOL;
    RobotOcupado : BOOL;
    RobotLivre : BOOL := TRUE;
    MaquinaLivre : BOOL := TRUE;
    BufferSaida : BOOL;

    (* INPUTS *)
    ADICIONARPECA : BOOL;
    REMOVERPECA : BOOL;

    (* virtual net *)
    PLACES_TEMP : ARRAY [1..7] OF PLACE := PLACES;
    TRANSITIONS_TEMP : ARRAY [1..6] OF TRANSITION := TRANSITIONS;

END_VAR


FUNCTION_BLOCK ADD_TOKENS
VAR_INPUT
   place  : UINT;
   tokens : UINT;
END_VAR

PLACES_TEMP[place].Marking := PLACES_TEMP[place].Marking + tokens;

END_FUNCTION_BLOCK


FUNCTION_BLOCK REMOVE_TOKENS
VAR_INPUT
   place : UINT;
   tokens : UINT;
END_VAR

PLACES_TEMP[place].Marking := PLACES_TEMP[place].Marking - tokens;
PLACES[place].Marking := PLACES[place].Marking - tokens;
```

```
END_FUNCTION_BLOCK


FUNCTION_BLOCK FIRE_TRANSITION
VAR_INPUT
   transition      : UINT;
END_VAR
VAR
   UPDATE_OUTPUTS  : UPDATE_OUTPUTS;
   REMOVE_TOKENS   : REMOVE_TOKENS;
   ADD_TOKENS      : ADD_TOKENS;
   counter         : UINT := 1;
   place           : UINT;
   tokens          : UINT;
   source_len      : UINT;
   target_len      : UINT;
END_VAR

(* removes tokens from input places *)
source_len := TRANSITIONS[transition].Source_len;
FOR counter := 1 TO source_len DO
   place := TRANSITIONS[transition].Source[counter];
   tokens := TRANSITIONS[transition].Source_weight[counter];
   REMOVE_TOKENS(place := place, tokens := tokens);
   UPDATE_OUTPUTS(place := place);
END_FOR

(* adds tokens to the output places *)
target_len := TRANSITIONS[transition].Target_len;
FOR counter := 1 TO target_len DO
   place := TRANSITIONS[transition].Target[counter];
   tokens := TRANSITIONS[transition].Target_weight[counter];
   ADD_TOKENS(place := place, tokens := tokens);
   UPDATE_OUTPUTS(place := place);
END_FOR

END_FUNCTION_BLOCK


FUNCTION_BLOCK IS_READY_TO_FIRE
VAR_INPUT
    transition  : UINT;
END_VAR
VAR_OUTPUT
    ready : BOOL;
END_VAR
VAR
```

```
        TIMER_OF_TRANSITION : TIMER_OF_TRANSITION;
        CONDITION_OF_TRANSITION : CONDITION_OF_TRANSITION;
        I: UINT;
        K: UINT;
        result : BOOL;
        IN : BOOL;
    END_VAR

    IN := FALSE;
    CONDITION_OF_TRANSITION(transition   :=   transition,   result   =>
result);
    IF result THEN
        (* checks whether all input places have enough tokens *)
        K := TRANSITIONS[transition].Source_len;
        FOR I := 1 TO K DO
                IF  PLACES[TRANSITIONS[transition].Source[I]].Marking <
TRANSITIONS[transition].Source_weight[I] THEN
                TIMER_OF_TRANSITION(transition := transition, IN := IN,
Q => ready);
                RETURN;
            END_IF
        END_FOR
        IN := TRUE;
    END_IF

    TIMER_OF_TRANSITION(transition  :=  transition,  IN  :=  IN,  Q  =>
ready);

    (* resets the timer if it already reached the desired value*)
    IFready = TRUE THEN
        TIMER_OF_TRANSITION(transition := transition, IN := FALSE);
    END_IF

    END_FUNCTION_BLOCK


    FUNCTION_BLOCK CONDITION_OF_TRANSITION
    VAR_INPUT
       transition : UINT;
    END_VAR
    VAR_OUTPUT
       result : BOOL;
    END_VAR
    VAR
      RE_ADICIONARPECA_1 : RE_TRIG;
      RE_REMOVERPECA_6 : RE_TRIG;
    END_VAR
```

```
    CASE transition OF
      1:
            RE_ADICIONARPECA_1(CLK := ADICIONARPECA);
            result := RE_ADICIONARPECA_1.Q ;
      2:
            result := TRUE ;
      3:
            result := TRUE ;
      4:
            result := TRUE ;
      5:
            result := TRUE ;
      6:
            RE_REMOVERPECA_6(CLK := REMOVERPECA);
            result := RE_REMOVERPECA_6.Q ;
       ELSE
           result := FALSE;
    END_CASE

    END_FUNCTION_BLOCK


    FUNCTION_BLOCK UPDATE_OUTPUTS
    VAR_INPUT
        place : UINT;
    END_VAR
    VAR
    END_VAR

    CASE place OF
      1:
            BUFFERENTRADA := PLACES_TEMP[1].Marking > 0;
      2:
            ROBOTOCUPADO    :=    PLACES_TEMP[2].Marking    >    0    OR
PLACES_TEMP[4].Marking > 0;
      3:
            MAQUINAOCUPADA := PLACES_TEMP[3].Marking > 0;
      4:
            ROBOTOCUPADO    :=    PLACES_TEMP[2].Marking    >    0    OR
PLACES_TEMP[4].Marking > 0;
      5:
            ROBOTLIVRE := PLACES_TEMP[5].Marking > 0;
      6:
            MAQUINALIVRE := PLACES_TEMP[6].Marking > 0;
      7:
            BUFFERSAIDA := PLACES_TEMP[7].Marking > 0;
```

```
ELSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK TIMER_OF_TRANSITION
VAR_INPUT
    transition : UINT;
    IN : BOOL;
END_VAR
VAR_OUTPUT
    Q : BOOL;
END_VAR
VAR
  TON_3 : TON;
  TON_4 : TON;
  TON_5 : TON;
END_VAR

CASE transition OF
  1:    Q := IN;
  2:    Q := IN;
  3:    TON_3(IN := IN, PT := T#10S);
        Q := TON_3.Q;
  4:    TON_4(IN := IN, PT := T#5S);
        Q := TON_4.Q;
  5:    TON_5(IN := IN, PT := T#30S);
        Q := TON_5.Q;
  6:    Q := IN;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK RE_TRIG
VAR_INPUT  CLK: BOOL; END_VAR
VAR_OUTPUT  Q: BOOL; END_VAR
VAR M: BOOL := FALSE; END_VAR

Q := CLK AND NOT M;
M := CLK;
END_FUNCTION_BLOCK


PROGRAM MAIN_PROGRAM
VAR
       IS_READY_TO_FIRE    : IS_READY_TO_FIRE;
       FIRE_TRANSITION     : FIRE_TRANSITION;
```

```
       transition          : UINT := 1;
       ready               : BOOL;
END_VAR

FOR transition := 1 TO TRANS_LENGTH DO
   IS_READY_TO_FIRE(transition := transition, ready => ready);
   IF ready THEN
       FIRE_TRANSITION(transition := transition);
   END_IF
END_FOR

(* update net *)
TRANSITIONS := TRANSITIONS_TEMP;
PLACES := PLACES_TEMP;

END_PROGRAM
```

# Anexo B: Tradução de um grafcet especificado em GML para código ST

O presente anexo, apresenta um grafcet especificado em GML e o correspondente código PLC gerado pela aplicação WEBGRAF 2.

## B.1 Grafcet de ilustração

A figura seguinte ilustra o grafcet a servir de exemplo:

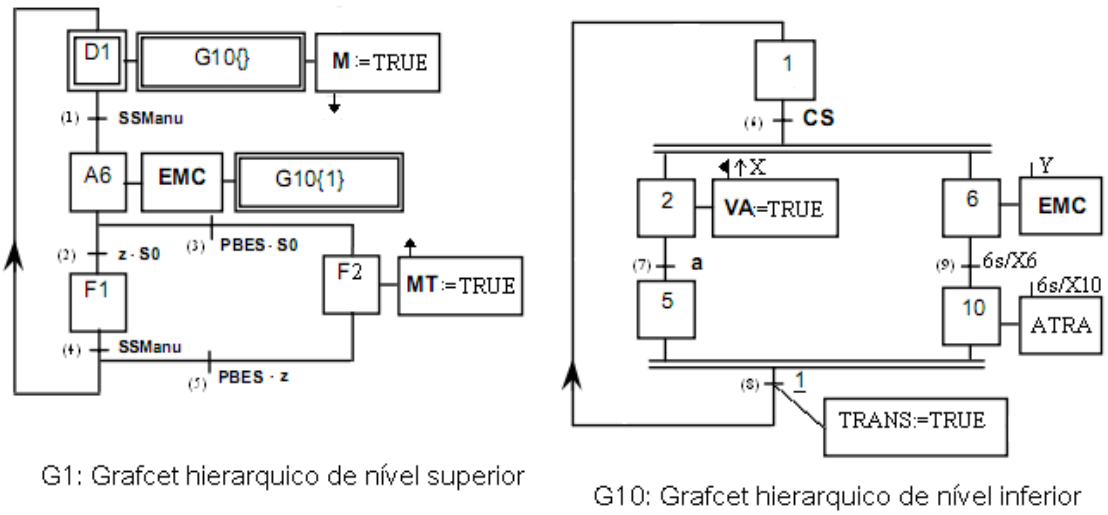



Figura B.1: Grafcet hierárquico.

## B.2 Especificação do modelo em GML

```
<gml>
  <localVars name="vmbvmvb - Local variables">
  <variable name="SSManu">
    <type>
        <BOOL />
    </type>
    <initialValue>
        <simpleValue value="FALSE" />
    </initialValue>
  </variable>
  <variable name="TRANS">
    <type>
        <BOOL />
    </type>
    <initialValue>
        <simpleValue value="FALSE" />
    </initialValue>
  </variable>
  <variable name="Z">
    <type>
        <BOOL />
    </type>
    <initialValue>
        <simpleValue value="FALSE" />
    </initialValue>
  </variable>
  <variable name="S0">
    <type>
        <BOOL />
    </type>
    <initialValue>
        <simpleValue value="False" />
    </initialValue>
  </variable>
  <variable name="PBES">
    <type>
        <BOOL />
    </type>
    <initialValue>
        <simpleValue value="False" />
    </initialValue>
  </variable>
  <variable name="X">
```

```
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="Y">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="A">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="CS">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="EMC">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="MT">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
```

```
<variable name="ATRA">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="M">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
<variable name="VA">
  <type>
      <BOOL />
  </type>
  <initialValue>
      <simpleValue value="False" />
  </initialValue>
</variable>
</localVars>

<!-- PARTIGAL GRAFCET G1 -->
<partialGrafcet localId="pg1" name="G1">

      <!-- STEPS -->
      <step name="D1" localId="1" initialStep="true">
        <position x="186" y="90" />
        <connectionPointIn>
            <connection refLocalId="21" />
        </connectionPointIn>
      </step>
      <actionBlock>
        <position x="252" y="96" />
        <connectionPointIn>
            <connection refLocalId="1" />
        </connectionPointIn>
        <action qualifier="onDeactivation">
            <reference name="M := TRUE" />
        </action>
      </actionBlock>
      <forcingOrder>
        <connectionPointIn>
```

```
      <connection refLocalId="1" />
  </connectionPointIn>
  <situation>
      <reference name="G10{}" />
  </situation>
</forcingOrder>
<step name="A6" localId="2" initialStep="false">
  <position x="186" y="202" />
  <connectionPointIn>
      <connection refLocalId="3" />
  </connectionPointIn>
</step>
<actionBlock>
  <position x="252" y="208" />
  <connectionPointIn>
      <connection refLocalId="2" />
  </connectionPointIn>
  <action qualifier="continuous">
      <reference name="EMC" />
  </action>
</actionBlock>
<forcingOrder>
  <connectionPointIn>
      <connection refLocalId="2" />
  </connectionPointIn>
  <situation qualifier="onDeactivation">
      <reference name="G10{1}" />
  </situation>
</forcingOrder>
<step name="F1" localId="4" initialStep="false">
  <position x="186" y="322" />
  <connectionPointIn>
      <connection refLocalId="5" />
  </connectionPointIn>
</step>
<step name="F2" localId="6" initialStep="false">
  <position x="346" y="322" />
  <connectionPointIn>
      <connection refLocalId="7" />
  </connectionPointIn>
</step>
<actionBlock>
  <position x="412" y="328" />
  <connectionPointIn>
      <connection refLocalId="6" />
  </connectionPointIn>
  <action qualifier="onActivation">
```

```
      <reference name="MT := TRUE" />
  </action>
</actionBlock>

<!-- TRANSITIONS -->
<transition height="44" width="44" localId="3">
  <position x="204" y="168" />
  <connectionPointIn>
      <connection refLocalId="1" />
  </connectionPointIn>
  <condition>
      <reference name="SSManu" />
  </condition>
</transition>
<transition height="44" width="44" localId="5">
  <position x="204" y="284" />
  <connectionPointIn>
      <connection refLocalId="22" />
  </connectionPointIn>
  <condition>
      <reference name="Z.S0" />
  </condition>
</transition>
<transition height="44" width="44" localId="7">
  <position x="276" y="284" />
  <connectionPointIn>
      <connection refLocalId="22" />
  </connectionPointIn>
  <condition>
      <reference name="PBES.S0" />
  </condition>
</transition>
<transition height="44" width="44" localId="8">
  <position x="8" y="228" />
  <connectionPointIn>
      <connection refLocalId="4" />
  </connectionPointIn>
  <condition>
      <reference name="SSManu" />
  </condition>
</transition>
<transition height="44" width="44" localId="9">
  <position x="100" y="228" />
  <connectionPointIn>
      <connection refLocalId="6" />
  </connectionPointIn>
  <condition>
```

```
            <reference name="PBES.Z" />
          </condition>
        </transition>

        <selectionConvergence localId="21">
          <position x="0" y="0" />
          <connectionPointIn>
              <connection refLocalId="8" />
              <connection refLocalId="9" />
          </connectionPointIn>
        </selectionConvergence>

        <selectionDivergence localId="22">
          <position x="0" y="0" />
          <connectionPointIn>
              <connection refLocalId="2" />
          </connectionPointIn>
        </selectionDivergence>
</partialGrafcet>

<!-- PARTIGAL GRAFCET G10 -->
<partialGrafcet localId="pg2" name="G10">
        <step name="1" localId="10" initialStep="false">
          <position x="706" y="70" />
          <connectionPointIn>
              <connection refLocalId="17" />
          </connectionPointIn>
        </step>
        <step name="2" localId="11" initialStep="false">
          <position x="626" y="198" />
          <connectionPointIn>
              <connection refLocalId="19" />
          </connectionPointIn>
        </step>
        <actionBlock>
          <position x="692" y="204" />
          <connectionPointIn>
              <connection refLocalId="11" />
          </connectionPointIn>
          <action qualifier="event">
              <reference name="VA := TRUE" />
              <condition name="re#X" />
          </action>
        </actionBlock>
        <step name="5" localId="12" initialStep="false">
          <position x="626" y="290" />
          <connectionPointIn>
```

```
      <connection refLocalId="16" />
  </connectionPointIn>
</step>
<step name="6" localId="13" initialStep="false">
  <position x="854" y="194" />
  <connectionPointIn>
      <connection refLocalId="19" />
  </connectionPointIn>
</step>
<actionBlock>
  <position x="920" y="200" />
  <connectionPointIn>
      <connection refLocalId="13" />
  </connectionPointIn>
  <action qualifier="continuous">
      <reference name="EMC" />
      <condition name="Y" />
  </action>
</actionBlock>
<step name="10" localId="14" initialStep="false">
  <position x="854" y="294" />
  <connectionPointIn>
      <connection refLocalId="18" />
  </connectionPointIn>
</step>
<actionBlock>
  <position x="920" y="300" />
  <connectionPointIn>
      <connection refLocalId="14" />
  </connectionPointIn>
  <action qualifier="continuous">
      <reference name="ATRA" />
      <condition name="6s/X10" />
  </action>
</actionBlock>

<!-- TRANSITIONS -->
<transition height="44" width="44" localId="15">
  <position x="724" y="144" />
  <connectionPointIn>
      <connection refLocalId="10" />
  </connectionPointIn>
  <condition>
      <reference name="CS" />
  </condition>
</transition>
<transition height="44" width="44" localId="16">
```

```
  <position x="644" y="264" />
  <connectionPointIn>
      <connection refLocalId="11" />
  </connectionPointIn>
  <condition>
      <reference name="A" />
  </condition>
</transition>
<transition height="44" width="44" localId="17">
  <position x="556" y="216" />
  <connectionPointIn>
      <connection refLocalId="20" />
  </connectionPointIn>
  <condition>
      <reference name="1" />
  </condition>
</transition>
<actionBlock>
  <position x="300" y="300" />
  <connectionPointIn>
      <connection refLocalId="17" />
  </connectionPointIn>
  <action qualifier="clearing">
      <reference name="TRANS:=TRUE" />
  </action>
</actionBlock>
<transition height="44" width="44" localId="18">
  <position x="872" y="264" />
  <connectionPointIn>
      <connection refLocalId="13" />
  </connectionPointIn>
  <condition>
      <reference name="6s/X6" />
  </condition>
</transition>

<simultaneousConvergence localId="20">
  <position x="0" y="0" />
  <connectionPointIn>
      <connection refLocalId="12" />
      <connection refLocalId="14" />
  </connectionPointIn>
</simultaneousConvergence>

<simultaneousDivergence localId="19">
  <position x="0" y="0" />
  <connectionPointIn>
```

```
            <connection refLocalId="15" />
          </connectionPointIn>
        </simultaneousDivergence>

  </partialGrafcet>
</gml>
```

## B.3 Código ST gerado a partir do modelo em GML

```
VAR_GLOBAL

(* STEPS *)
XD1 : BOOL := TRUE;
XA6 : BOOL;
XF1 : BOOL;
XF2 : BOOL;
X1 : BOOL;
X2 : BOOL;
X5 : BOOL;
X6 : BOOL;
X10 : BOOL;

(* VIRTUAL STEPS *)
XD1_TEMP : BOOL := TRUE;
XA6_TEMP : BOOL;
XF1_TEMP : BOOL;
XF2_TEMP : BOOL;
X1_TEMP : BOOL;
X2_TEMP : BOOL;
X5_TEMP : BOOL;
X6_TEMP : BOOL;
X10_TEMP : BOOL;

TRANSITION_LENGTH : UINT := 9;

(* OUTPUT VARIABLES *)
SSManu : BOOL;
TRANS : BOOL;
Z : BOOL;
S0 : BOOL;
PBES : BOOL;
X : BOOL;
Y : BOOL;
A : BOOL;
CS : BOOL;
```

```
EMC : BOOL;
MT : BOOL;
ATRA : BOOL;
M : BOOL;
VA : BOOL;

END_VAR


FUNCTION_BLOCK CHECK_CONDITION_OF_TRANSITION
VAR_INPUT
  TRANSITION : UINT;
END_VAR
VAR_OUTPUT
  CONDITION : BOOL;
END_VAR
VAR
  TON_9 : TON;
END_VAR
CASE TRANSITION OF
  1:
        CONDITION := SSManu;
  2:
        CONDITION := Z AND S0;
  3:
        CONDITION := PBES AND S0;
  4:
        CONDITION := SSManu;
  5:
        CONDITION := PBES AND Z;
  6:
        CONDITION := CS;
  7:
        CONDITION := A;
  8:
        CONDITION := TRUE;
  9:
        TON_9(IN := X6, PT := T#6s);
        CONDITION := TON_9.Q;
        IF CONDITION THEN
              TON_9(IN := FALSE);
        END_IF
  ELSE
        CONDITION := FALSE;
END_CASE
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK CHECK_IF_INPUT_STEPS_ARE_ACTIVE
VAR_INPUT
  TRANSITION: UINT;
END_VAR
VAR_OUTPUT
  INPUT_STEPS_ACTIVE: BOOL;
END_VAR
VAR
END_VAR
CASE TRANSITION OF
  1:
        INPUT_STEPS_ACTIVE := XD1;
  2:
        INPUT_STEPS_ACTIVE := XA6;
  3:
        INPUT_STEPS_ACTIVE := XA6;
  4:
        INPUT_STEPS_ACTIVE := XF1;
  5:
        INPUT_STEPS_ACTIVE := XF2;
  6:
        INPUT_STEPS_ACTIVE := X1;
  7:
        INPUT_STEPS_ACTIVE := X2;
  8:
        INPUT_STEPS_ACTIVE := X5 AND X10;
  9:
        INPUT_STEPS_ACTIVE := X6;
  ELSE
      INPUT_STEPS_ACTIVE := FALSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK ENABLE_STEP
VAR_INPUT
  STEP:UINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
CASE STEP OF
  1:
      XD1_TEMP:=TRUE;
  2:
```

```
        XA6_TEMP:=TRUE;
  3:
        XF1_TEMP:=TRUE;
  4:
        XF2_TEMP:=TRUE;
  5:
        X1_TEMP:=TRUE;
  6:
        X2_TEMP:=TRUE;
  7:
        X5_TEMP:=TRUE;
  8:
        X6_TEMP:=TRUE;
  9:
        X10_TEMP:=TRUE;
  ELSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK DISABLE_STEP
VAR_INPUT
  STEP:UINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
CASE STEP OF
  1:
        XD1_TEMP:=FALSE;
  2:
        XA6_TEMP:=FALSE;
  3:
        XF1_TEMP:=FALSE;
  4:
        XF2_TEMP:=FALSE;
  5:
        X1_TEMP:=FALSE;
  6:
        X2_TEMP:=FALSE;
  7:
        X5_TEMP:=FALSE;
  8:
        X6_TEMP:=FALSE;
  9:
        X10_TEMP:=FALSE;
```

```
  ELSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_ACTIONS_OF_TRANSITION
VAR_INPUT
  TRANSITION : UINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR

CASE TRANSITION OF
  8:
        TRANS:=TRUE;
  ELSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_ACTIONS_ON_ACTIVATION
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
  RE_XF2 : RE_TRIG;
END_VAR

RE_XF2(CLK := XF2);
IF RE_XF2.Q THEN
  MT := TRUE;
END_IF
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_ACTIONS_ON_DEACTIVATION
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
  F_XD1 : F_TRIG;
END_VAR
```

```
F_XD1(CLK := XD1);
IF F_XD1.Q THEN
  M := TRUE;
END_IF
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_ACTIONS_ON_EVENT
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
RE_X_1 : RE_TRIG;
END_VAR

RE_X_1(CLK := X);
IF X2 AND (RE_X_1.Q) THEN
  VA := TRUE;
END_IF
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_CONTINUOUS_ACTIONS
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
  TON_2 : TON;
END_VAR

IF X10 THEN
  TON_2(IN := X10, PT := T#6s);
ELSE
  TON_2(IN := FALSE);
END_IF
  ATRA := TON_2.Q;
  EMC := XA6 OR X6 AND Y;
END_FUNCTION_BLOCK


FUNCTION_BLOCK EXECUTE_FORCING_ORDERS
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
```

```
VAR
END_VAR

IF XD1 THEN
  X1_TEMP := FALSE;
  X2_TEMP := FALSE;
  X5_TEMP := FALSE;
  X6_TEMP := FALSE;
  X10_TEMP := FALSE;
END_IF
IF XA6 THEN
  X1_TEMP := TRUE;
  X2_TEMP :=FALSE;
  X5_TEMP :=FALSE;
  X6_TEMP :=FALSE;
  X10_TEMP :=FALSE;
END_IF
END_FUNCTION_BLOCK


FUNCTION_BLOCK IS_STEP_ACTIVE
VAR_INPUT
  STEP:UINT;
END_VAR
VAR_OUTPUT
  ACTIVE:BOOL;
END_VAR
VAR
END_VAR

CASE STEP OF
  1:
        ACTIVE := XD1;
  2:
        ACTIVE := XA6;
  3:
        ACTIVE := XF1;
  4:
        ACTIVE := XF2;
  5:
        ACTIVE := X1;
  6:
        ACTIVE := X2;
  7:
        ACTIVE := X5;
  8:
        ACTIVE := X6;
```

```
  9:
        ACTIVE := X10;
  ELSE;
END_CASE
END_FUNCTION_BLOCK


FUNCTION_BLOCK FIRE_TRANSITION
VAR_INPUT
  TRANSITION : UINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
  ENABLE_STEP : ENABLE_STEP;
  DISABLE_STEP : DISABLE_STEP;
END_VAR
CASE TRANSITION OF
  1:
        DISABLE_STEP(STEP := 1);
        ENABLE_STEP(STEP := 2);
  2:
        DISABLE_STEP(STEP := 2);
        ENABLE_STEP(STEP := 3);
  3:
        DISABLE_STEP(STEP := 2);
        ENABLE_STEP(STEP := 4);
  4:
        DISABLE_STEP(STEP := 3);
        ENABLE_STEP(STEP := 1);
  5:
        DISABLE_STEP(STEP := 4);
        ENABLE_STEP(STEP := 1);
  6:
        DISABLE_STEP(STEP := 5);
        ENABLE_STEP(STEP := 6);
        ENABLE_STEP(STEP := 8);
  7:
        DISABLE_STEP(STEP := 6);
        ENABLE_STEP(STEP := 7);
  8:
        DISABLE_STEP(STEP := 7);
        DISABLE_STEP(STEP := 9);
        ENABLE_STEP(STEP := 5);
  9:
        DISABLE_STEP(STEP := 8);
        ENABLE_STEP(STEP := 9);
```

```
    END_CASE
    END_FUNCTION_BLOCK


    PROGRAM MAIN_PROGRAM
    VAR
      CHECK_IF_INPUT_STEPS_ARE_ACTIVE                                    :
CHECK_IF_INPUT_STEPS_ARE_ACTIVE;
      CHECK_CONDITION_OF_TRANSITION : CHECK_CONDITION_OF_TRANSITION;
      EXECUTE_CONTINUOUS_ACTIONS : EXECUTE_CONTINUOUS_ACTIONS;
      EXECUTE_ACTIONS_OF_TRANSITION : EXECUTE_ACTIONS_OF_TRANSITION;
      EXECUTE_ACTIONS_ON_ACTIVATION : EXECUTE_ACTIONS_ON_ACTIVATION;
      EXECUTE_ACTIONS_ON_DEACTIVATION                                    :
EXECUTE_ACTIONS_ON_DEACTIVATION;
      EXECUTE_ACTIONS_ON_EVENT : EXECUTE_ACTIONS_ON_EVENT;
      EXECUTE_FORCING_ORDERS : EXECUTE_FORCING_ORDERS;
      FIRE_TRANSITION : FIRE_TRANSITION;

      IS_STEP_ACTIVE : IS_STEP_ACTIVE;
      ENABLE_STEP : ENABLE_STEP;
      DISABLE_STEP: DISABLE_STEP;

      INPUT_STEPS_ACTIVE : BOOL;
      CONDITION : BOOL;
      TRANSITION : UINT;
      FIRSTCYCLE : BOOL := TRUE;
      OUTPUT_TRANSITION_ENABLED : BOOL;
      INPUT_TRANSITION_ENABLED : BOOL;
    END_VAR

    IF FIRSTCYCLE THEN
      EXECUTE_FORCING_ORDERS();
      EXECUTE_CONTINUOUS_ACTIONS();
      FIRSTCYCLE := FALSE;
    END_IF

    (* for each transition *)
    FOR TRANSITION := 1 TO TRANSITION_LENGTH DO
      CHECK_IF_INPUT_STEPS_ARE_ACTIVE(TRANSITION     :=     TRANSITION,
INPUT_STEPS_ACTIVE => INPUT_STEPS_ACTIVE);
      CHECK_CONDITION_OF_TRANSITION(TRANSITION      :=      TRANSITION,
CONDITION => CONDITION);
      IF INPUT_STEPS_ACTIVE AND CONDITION THEN
            FIRE_TRANSITION(TRANSITION := TRANSITION);
            EXECUTE_ACTIONS_OF_TRANSITION(TRANSITION := TRANSITION);
      END_IF
    END_FOR
```

```
(* EXECUTE ACTIONS *)
EXECUTE_CONTINUOUS_ACTIONS();
EXECUTE_ACTIONS_ON_EVENT();
EXECUTE_ACTIONS_ON_ACTIVATION();
EXECUTE_ACTIONS_ON_DEACTIVATION();
EXECUTE_FORCING_ORDERS();

(* update net *)
XD1:=XD1_TEMP;
XA6:=XA6_TEMP;
XF1:=XF1_TEMP;
XF2:=XF2_TEMP;
X1:=X1_TEMP;
X2:=X2_TEMP;
X6:=X6_TEMP;
X5:=X5_TEMP;
X10:=X10_TEMP;

END_PROGRAM


FUNCTION_BLOCK RE_TRIG
VAR_INPUT  CLK: BOOL; END_VAR
VAR_OUTPUT  Q: BOOL; END_VAR
VAR M: BOOL := FALSE; END_VAR

Q := CLK AND NOT M;
M := CLK;
END_FUNCTION_BLOCK
```